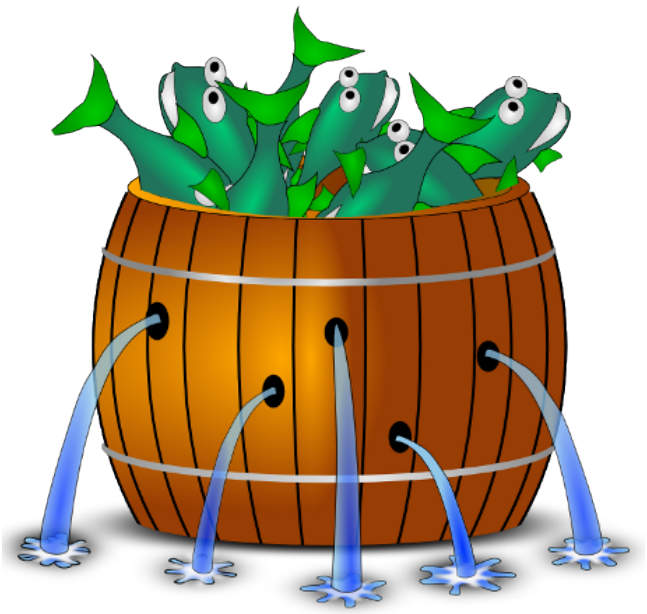


Barreldish OS

AOS Project Report - Spring Semester 2022

Bowen Wu, Jasmin Schult, Matteo Oldani, Sven Grübel

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Contents

1 Introduction **5**
- 1.1 Structure . . . 5
- 1.2 Extra challenges . . . 5

2 Memory Allocator **7**
- 2.1 Data structure . . . 7
- 2.2 Coalescing . . . 8
- 2.3 Allocation of book-keeping memory . . . 9
- 2.4 Error handling . . . 9
- 2.5 A separate domain for the memory manager . . . 9

3 Pager **11**
- 3.1 Data structures . . . 11
  - 3.1.1 Tracking of page tables and mappings . . . 11
  - 3.1.2 Tracking of virtual address space allocation . . . 12
- 3.2 Refills . . . 13
- 3.3 Passing the paging state to the child . . . 14
  - 3.3.1 Passing capabilities . . . 14
  - 3.3.2 Passing the data structure . . . 15
- 3.4 How multithread support solved the refill issues . . . 16
- 3.5 Page fault handling . . . 16

4 Multi-threads **17**
- 4.1 Challenge of multi-thread support . . . 17
  - 4.1.1 Data structures . . . 17
  - 4.1.2 Refilling . . . 17
- 4.2 Overview of our locking approach . . . 17
  - 4.2.1 Why not just a big, global lock? . . . 17
  - 4.2.2 Locking invariant . . . 19
- 4.3 Data structure adjustments . . . 19
- 4.4 The problem of unbacked stacks! . . . 19
- 4.5 Page fault handling revisited . . . 20

5 Processes **22**
- 5.1 Introduction . . . 22
- 5.2 Our implementation . . . 22
  - 5.2.1 Load the binary and cmd line . . . 22
  - 5.2.2 Find and map ELF . . . 23

5.2.3 C-Space and V-Space creation . . . 23
5.2.4 Parsing the ELF . . . 23
5.2.5 The last steps . . . 24
5.2.6 Processes management . . . 24
5.3 Spawn Server . . . 25
5.3.1 Spawn server struggles . . . 25
5.4 Spawning order . . . 26
5.5 Performance Analysis . . . 27
5.6 Conclusion . . . 27

6 Multi-core Support **30**
6.1 Introduction . . . 30
6.2 Memory division among cores . . . 30
6.2.1 The first approach . . . 30
6.2.2 The second approach . . . 31
6.3 Passing the multiboot info . . . 31
6.3.1 Signalling core readiness to use UMP . . . 32

7 Remote Procedure Calls **33**
7.1 Overview . . . 33
7.2 Naming . . . 34
7.2.1 RPC client interface . . . 35
7.3 Binding and Connecting . . . 35
7.3.1 Infrequent communication . . . 35
7.4 RPC Message abstraction . . . 36
7.4.1 Capability representation . . . 36
7.4.2 The Header . . . 37
7.4.3 The payload . . . 37
7.5 Routing and channel selection . . . 38
7.6 The RPC Channel abstraction . . . 38
7.6.1 At first, there was LMP . . . 39
7.6.2 Then, there came UMP . . . 40
7.6.3 Take 2: the final, non-blocking solution . . . 41
7.7 Message Manager overview . . . 45
7.7.1 Typical rpc operations . . . 45
7.7.2 Specialized wrappers around send and receive . . . 45
7.7.3 Mailboxing . . . 45
7.7.4 Round-robin polling . . . 46
7.7.5 Decoupling . . . 46
7.8 Lower level channel operations . . . 47
7.8.1 LMP message marshalling . . . 47
7.8.2 Large LMP message handling . . . 47
7.8.3 UMP send and receive . . . 47
7.9 Limitations . . . 48

8 Shell **50**
8.1 Introduction . . . 50
8.2 UART in user space . . . 50
8.2.1 Preparing the UART . . . 51
8.2.2 The client perspective . . . 51

8.2.3 The protocol . . . 51
8.2.4 UART Perspective . . . 55
8.2.5 Client perspective on Protocol . . . 55
8.2.6 Comments on the design choices . . . 55
8.2.7 Drawbacks and compromises . . . 56
8.3 The Shell . . . 56
8.3.1 The input system . . . 56
8.3.2 Command execution . . . 57
8.3.3 Well-known Commands . . . 57
8.3.4 Network commands . . . 59
8.3.5 How to add new commands . . . 60
8.3.6 Blocking spawning . . . 60
8.3.7 Outside the Well-Known commands . . . 61
8.3.8 Performance Analysis . . . 61
8.4 Conclusion . . . 62

9 Capabilities Revisited **64**
9.1 Correctness . . . 64
9.1.1 Correct System State . . . 64
9.1.2 Operational Consistency . . . 65
9.1.3 Supported operations, restrictions and requirements . . . 66
9.1.4 Why supporting many cores is hard . . . 69
9.2 Design process or TLA+ to the rescue! . . . 72
9.3 The protocol . . . 72
9.3.1 Send capability . . . 73
9.3.2 Send capability with ownership transfer . . . 73
9.3.3 Revoke . . . 74
9.3.4 Delete Last . . . 76
9.3.5 Retype . . . 77
9.4 Implementation . . . 79
9.5 The TLA+ model . . . 80
9.5.1 limitations and restrictions . . . 80
9.5.2 Aspects that are faithfully represented . . . 81
9.6 Model checking . . . 82
9.7 RAM reclamation . . . 83
9.7.1 Why the existing mechanism did not work . . . 83
9.7.2 How to fix it . . . 84
9.7.3 Limitations . . . 85
9.8 Changes to the Kernel . . . 86

10 File System **87**
10.1 Introduction . . . 87
10.2 Architecture . . . 88
10.2.1 Layer-1 SDHC driver . . . 88
10.2.2 Layer-2 Cache . . . 90
10.2.3 Layer-3 FAT-32 standard . . . 90
10.2.4 Layer-4 File system server . . . 91
10.2.5 Layer 5 and beyond . . . 92
10.3 Design decisions . . . 92
10.4 Loading ELF for program execution . . . 93

10.5 Efforts to improve the device driver . . . 94
10.6 Performance Evaluation . . . 95
10.7 Limitations . . . 96

11 Networking **97**
11.1 Architecture . . . 97
11.2 Design Choices . . . 97
11.3 Internal Functionality . . . 98
11.3.1 Interacting with the "enet" Driver . . . 99
11.3.2 Receiving Network Data . . . 99
11.3.3 Sending Network Data . . . 100
11.3.4 Keeping an ARP Cache . . . 101
11.3.5 Keeping a Port Registry . . . 102
11.4 External Functionality . . . 103
11.4.1 Listening to a Port . . . 103
11.4.2 Receiving Data . . . 104
11.4.3 Sending Data . . . 104
11.5 Putting it Together . . . 105
11.6 Performance Evaluation . . . 107
11.7 Limitations . . . 107

12 Conclusions **108**
12.1 Notes about Grading . . . 108
12.2 End of life . . . 108

A TLA+ Pretty Printed Specification **109**

Chapter 1

Introduction

In the following pages, you will go through all our achievements as well as all our struggles in building the OS. We tried to be consistent and coherent with our design decisions and with the spirit of the course. We hope that this report can explain them well. In any case, we enjoyed the journey and we hope that you can have a nice time reading our report.

1.1 Structure

We decided to diverge from the proposed template for the report because to explain the process on a milestone base, we would have to re-explain part of our implementation multiple times. To avoid repetitions and increase the clarity of the description of our OS we decided to divide the report by topic. However, we tried to present all the topics in an order that at least followed the milestones.

1.2 Extra challenges

In the following section you will find a list of all the extra challenges that we implemented, starting from the first group milestone:

- Milestone 2
  - Pass the paging state from the parent to the child process
  - Paging unmapping code
- Milestone 3
  - Large messages
- Milestone 4
  - Dynamic stack allocation
- Milestone 5
  - Booting all application cores of the platform
- Milestone 6

  - Sending large messages across cores
  - Performance measurements
  - Direct binding between processes is possible
- Filesystem
  - Optimizing the performance of the block driver
- Capabilities revisited
  - Optionally transferring ownership of the capability between cores

Chapter 2

Memory Allocator

2.1 Data structure

The memory manager's bookkeeping centres around the so-called `mem` structures, each of which describes a contiguous region of RAM (or resources if the objtype the manager manages is different from RAM). For this purpose, each mem has a `base_address` and `size member`.

All existing `mems` are part of the `neighbours` doubly-linked list which, as the name implies, connects adjacent `mems`. An invariant of this list is that it is always complete in the sense that all its `mems` taken together yield exactly the RAM represented by the initial capabilities added to the manager using `mm_add`, without any gaps or overlaps.

Each `mem` is either free or used as indicated by its boolean `free` member. Free means it is available for allocation, used means that the region has been handed out to client(s) which haven't yet freed their respective region(s).

Free `mems` are additionally part of the *free-tree*, a red-black tree sorted according to size, alignment and finally, to break ties, according to `base_address`. This tree is used to quickly find a suitable free block to answer an `mm_alloc` request:

1. search for a `mem` with the requested size and the alignment as base address (will find blocks with same alingment if existing): Try to find a perfect fit

2. search for a mem with size (requested_size + alignment) and base address `BASE_PAGE_SIZE` (not set to zero because the alignment of zero is infinite): find a block guaranteed to contain a subblock of suitable size and alignment

3. search through the entire neighbour list: find any regions *in-between* the two queries, i.e. of smaller size than 2 but still containing a suitably aligned subblock

Used `mems` are part of a *used-tree*, also a red black tree sorted according to base address. This structure is used to quickly find the relevant `mem` for (partial) free operations.

To enable the futher splitting of regions as needed, each `mem` maintains a pointer to the initial capability it was derived from in its `mother` member. For

free `mems`, a retype operation of their mother to a capability corresponding to them must always succeed. As a result, capabilities of free `mems` are created on demand to serve an alloc request but do not exist otherwise.

Figure 2.1 provides an illustration of the described data structures: the blue blobs represent the initial capabilities, red rectangles represent used regions while green rectangles represent free regions. Note that for simplicity, the trees and linked lists are drawn individually, but in truth every region object is present in both the neighbour linked list as well as the appropriate tree depending on its usage status.

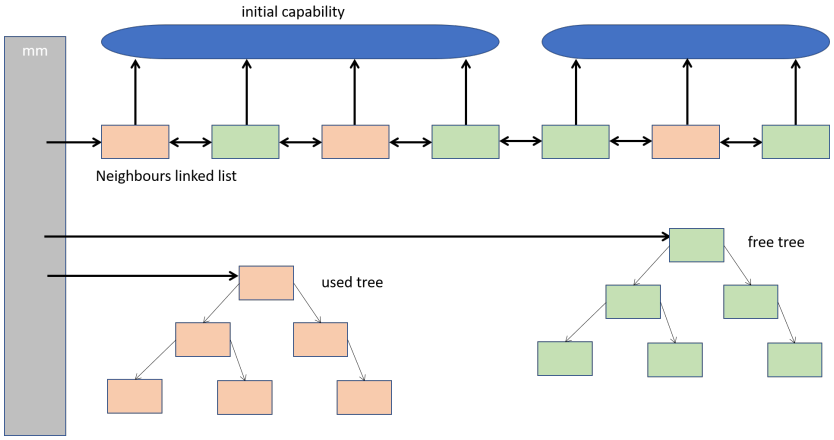


Figure 2.1: An overview of the memory server's data structures

2.2 Coalescing

Coalescing of `mems` is performed both for adjacent free and adjacent used blocks. This is one of the main differences to classical free-list allocators: they cannot coalesce used blocks because they cannot trust their clients to indicate the correct sizes to be freed. In the capability system this is not an issue because capabilities cannot be (accidentially) merged or forged by clients.

However, this design decision has implications for future operations: if the manager merges used regions, then how are we going to identify the memory that used to belong to a particular process to free it once said exits?

For now, we reap the performance benefits: the neighbour list is as a result at most 2x longer than a free-list only and as a result if we need to resort to searching through the entire list, the performance is at most 2x worse than for a regular free-list allocator. The additional invariant that the mems in the neighbour list must be alternatingly free and used if they have the same *mother*.

Naturally, no coalescing can take place across different *mothers* (i.e. initial capabilities) because the manager cannot merge them unless it somehow obtain their common ancestor (which should not happen because it would imply that the kernel is leaking its `base` capabilities). This is one of the advantages of using `mother` pointers instead of storing the capabilitity into the mems directly (apart from the memory footprint): it facilitates the comparison of mothers.

2.3 Allocation of book-keeping memory

Use two slab allocators: one for the `mems` and one for the mother caprefs (since an arbitrary number of initial free regions might be added, we cannot fix the required number of such mothers).

Additionally, the already provided slot allocator is used to allocate new capability slots into which free (sub) `mems` are retyped to hand out to the client in reponse to allocation requests.

Refills: Have determined the worst case chain of refills that can be triggered by all allocators in the memory manager as well as the pager slabs (see below) needing to be refilled simultaneously, which results in at most 24 calls to `mm_alloc`. Therefore, we keep 24 times the resources used by `mm_alloc` in the worst case in reserve.

- 2 slot allocator refills (mm's and the default one used by the pager: 2 `mm_alloc` calls (one each for new L1, L2 Cnode in the worst case).
- refilling of 5 slab allocators (mems, mothers and paging nodes, region nodes of the pager (see below) and default slot allocator slabs): For each, 1 call to `mm_alloc` for the frame to map and 3 calls for the required page tables to map the block.

2.4 Error handling

All of exposed methods of the memory manager try to perform the capability operations that can potentially fail first, before updating the book-keeping data structure. Ensures that data structure remains consistent even if certain operations fail.

For free operations, the memory manager checks if the capability can be retyped from its mother (see invariant above) after deleting it - detects if siblings or descendants still exist and can thus prevent future errors in capability retyping. However, this check is not ideal because it requires deleting the cap first and if the check fails because a descendant exists then there is no way to fully free the region afterwards because the client doesn't have the corresponding capability anymore (mm deleted it and can't reproduce it)! However, the drawbacks of this check are outweighed by its benefits in our opinion because it is better to fail at this stage and be unable to reuse the memory than fail to retype supposedly free memory later, which is an error from which the memory manager cannot easily recover because there is no straightforward way to tell which (partial) subregion is responsible for the retype failure.

2.5 A separate domain for the memory manager

During the process milestone, we have decided to migrate the mm into a dedicated domain, in order to provide better (fault) isolation and separation of our OS's core functionality.

In order to set up the mem server, init is walking through the regions of its own initial version of the memory manager, splitting every free region from its initial capability (since it could be that parts of the inital capability are used

by init and can therefore not be awarded to the mem server) and sending the resulting capability over the the mem server domain using the dedicated `mm_add` rpc call.

During this transfer, init is temporarily switching back to the fixed version of alloc (because the mm serving ram alloc calls while we are walking over the its list of regions is clearly a bad idea) and once complete it sets its ram alloc function to use the `get_ram_cap` rpc function to request memory at the mem server from now on.

Please note that init is NOT allowed to consult its own version of the memory manager anymore after this transfer, which is also why the monitor (which is initialized strictly after) will use the free rpc call to enable memory reclamation.

Chapter 3

Pager

3.1 Data structures

3.1.1 Tracking of page tables and mappings

The pager's book-keeping centres around so-called `paging_nodes`. Each of these nodes describes an existing page table mapping at the level indicated by the `level` member. The corresponding mapping is stored in the `mapping` capref member and the mapped page table or frame capability in the `table` member.

These paging nodes are arranged in a tree-of-trees (red black trees to be precise): each paging node is a member of the tree belonging to the higher level page table node it is mapped into (`level_tree member`) (while the 0-level page table itself is anchored in the paging state struct) and has a pointer to the tree containing the next-level page tables or frames that are mapped into itself (`next_tree` member). See Fig.3.1.

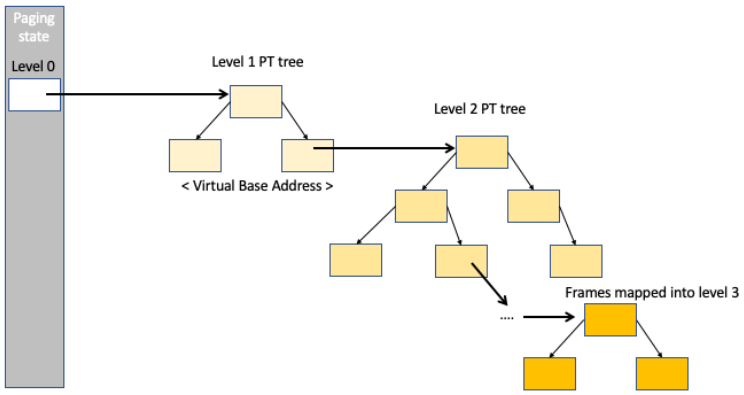


Figure 3.1: Trees that are used to track allocated virtual addresses in the pager state.

The main benefit of this approach is the reduced memory footprint in the case of mappings that are scattered over the the entire virtual address space, since we

are not explicitly storing all of the (mostly unused in this case) 512 slots of each page table. However, it certainly results in a noticeable performance overhead because in the worst case we need to chase 18 tree-pointers ($2 \cdot log_2(512)$) to find a specific "slot entry" in a page table, which results in bad cache locality. However, the hope is that the mapping operations are comparatively few (especially with respect to the number of page table walks performed on these mappings) so resolving the trade-off in favour of memory footprint is justifiable.

Note also that at first glance, a tree seems a bad choice for the last page table level since it is said that is most likely to be filled entirely by mapped-in frame pages - however, it is likely that such a (comparatively large) mapping of virtual address space is performed with few map calls on a few large frames, in which case we only need to generate one tree node for each mapping and can still benefit from the sparsity (because unlike the MMU we are not interested on mappings on a page-by-page level but only in the resulting page table and mapping capabilities to comply with future map or unmap operations).

3.1.2 Tracking of virtual address space allocation

While a very simple virtual address allocation that only maintains the virtual address from which on the entire space is "unused" and just allocates from there might still be feasible to solve milestone 2, it immediately breaks down once we have cycled through the entire address space once, which is intolerable in the case of processes that need to create and subsequently destroy lots of mappings (such as the "spawn server").

Our main realisation here was that the operations required to find, allocate and free virtual address space are almost identical to said of performed for physical memory in the mm (minus the capability operations). We have therefore factored out this management into a generic `region_allocator` data structure that can be parametrized with different coalescing and "region comparison" policies but uses the same free-tree, used-tree and neighbour-list structures as the original memory management to serve requests. See Fig.3.2 for a illustration of the region allocator.

The main difference between the region allocator used for physical and virtual address allocation is in the coalescing policy and an additional required operation to find a specific free region (to serve allocations of fixed virtual addresses):

Coalescing policy In contrast to the mm, we are not operating on capabilities that prove ownership of a precisely defined region of physical address space. As a result, we are not allowed to coalesce neighbouring used regions anymore because we cannot rely on the client to correctly indicate the size of the virtual address space region that should be freed. On the other hand, the initial virtual address space is now contiguous and therefore we don't have to take care to prevent coalescing across different initial mother capabilities anymore. To summarize, the coalescing policy is simply `false` and `true` for two neighbouring used and free regions respectively.

Find a specific free region This operation is not required for the mm because clients cannot wish for a specific physical address since it is opaque to the

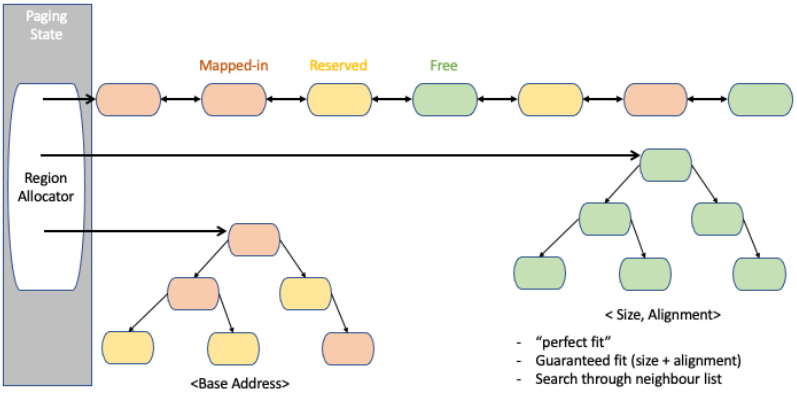


Figure 3.2: Region allocator used by the pager. It contains a list of memory regions in different states, a tree of free regions and a tree of allocated/reserved regions.

user anyway because of the virtual-physical address translation. Luckily, despite not being designed to support this operation, the original manager design can nevertheless do so efficiently: We simply search for the closest previous *used region* in our *used-tree* which allows us to instrument the found region as an index into the neighbour-list. The requested *free region* is then an immediate neighbour or only few hops away (in the case when the manager received several disjoint initial regions).

3.2 Refills

As with the mm, we also use the watermarking approach to avoid dead-end refills for the pager. However, there is an additional subtlety to consider in case of the pager: we need to try to avoid slab refills during the hierarchical mapping process. Imagine a slab refill is triggered by a request for the new page table capability that we require to map a specific virtual address. It could then happen that the refill itself maps in the page table we were trying to map because the virtual address chosen for the new slab frame also maps into this missing page table. This then causes our mapping process to fail with an "already mapped error".

The remedy is, we are keeping ready to map page tables in reserve, one each for levels 1 to 3. Whenever a client requests a new mapping or the pager must refill its slab allocators, we firstly refill these 3 reserve page tables before we initiate the mapping.

Despite this, it can happen that we need to perform a mapping with missing reserve page tables, for instance if the request for a new reserve page table capability triggers a slab refill in the memory manager. Luckily, such a mapping call caused by an mm refill can only happen once because the mm's refill is a no-op if it is triggered while it is refilling, so it is safe to request new page table capabilities during the mapping process in this case.

Or is it...? The remaining issue is that the default slot allocator is also in need of slabs for its book-keeping, a fact that was overlooked in the original pager design because of the (in hindsight somewhat stupid) assumption that the resources consumed by the default slot allocator are identical to the preallocated version. To make matters worse, there is no easy way for the pager to control the refills of the default slot allocator. So have our reserved page tables all been for nothing? No, because they limit the number of nested calls to the hierarchical mapping procedure to at most one. As a result, in the (quite unlikely) event that the default slot allocator must refill during the slab refill of the memory manager which was triggered by requesting a new reserve page table, we can detect this case by checking for the "already mapped" error, destroying the page table that we just requested (since likely the entire call chain was triggered by trying to refill this specific reserved page table slot and if we do not destroy it but use it as reserve it will simply be overwritten by the outer call and will therefore be lost forever) and again searching down the page table hierarchy to find the new closest parent page table in which we should try to map.

Refill threshold: same thoughts as above: at worst, we need to refill 5 slab allocators which require at most 4 `paging_nodes` each (3 for the level 1 to 3 page tables and a final one for the mapped-in frame) and thus it suffices to keep 24 paging nodes in reserve.

The ultimate solution for refilling is discussed in 4.2.

3.3 Passing the paging state to the child

This extra challenge was quite interesting to design and implement, which is why we dedicate an entire section to it.

The discussion of the passing mechanism can be split into two aspects: the handling of the capabilities as well as the handling of the entire data structure.

3.3.1 Passing capabilities

The layout of the CSpace has a dedicated L2 Cnode for this purpose called pagecn. Passing the vspace capabilities to the child cspace was therefore a simple matter of traversing the `paging_node` tree-of-trees and using `cap_copy` to copy each capability into the next free slot in that specific CNode. An immediate limitation of this approach is that no more than 256 such capabilities can be passed like this unless we allocate additional L2 Cnodes for this purpose. Since the L1 Cnode layout is fixed at compile time by the constants defined in `init.h` (see the USER offset), this is an issue that as of now cannot easily be addressed on-demand when a large process with many elf sections is spawned and will thus require further investigation.

Additionally, we found that the mapping capabilities seem to capture the specific page table capability into which they map (rather than referring to the true page table object itself) because the deletion of the parent's page table capabilities causes segfaults in the child - while there certainly is a rationale behind this capability design decision, we were still somewhat surprised by it because it implies that we can neither recover the slot the the page table uses in the parent once the child is constructed, nor can we ever free the ram used by the page table for good because the parent's copy will still always exist (unless

we keep it around in some structure and delete the copy once the child exits or we implement the revocation of capabilities).

Restoring the capabilities once we are executing in the child is simply the reverse procedure - we walk through the tree-of-trees in the same order while reading out the capabilities from the pagecn cnode and storing them into our paging data structure. A noteworthy hiccup in the process of implementing this step was the assumption (again, in hindsight mostly stupid since that would require a global way of specifying cnodes which is inherently unsafe because then any process could attempt to copy "useless" capabilities into the slots of another unsuspecting process and thus mount a "new slot denial of service attack" on the other process) that the reference to a future child cnode from the perspective of the parent would be identical to the reference of the corresponding cnode from the perspective of the child - which resulted in an attempt to update the data structure with newly-copied into capability references during the copying step in the parent (which of course caused the subsequent validation of the capabilities in the child to fail miserably :-)).

3.3.2 Passing the data structure

The main realisation that enabled this step to run smoothly was that apart from the `paging_state` struct itself, all the memory our pager is using for book-keeping is acquired and allocated by its two slab allocators. As a result, we can easily pass the entire structure to the child by:

1. Allocating a *pager frame* when initializing the child's pager in the parent (using `paging_init_foreign`), mapping it into the parent vspace and using the resulting backed virtual address region to grow the child's slab allocators while leaving space in the beginning to later store the `paging_state` struct itself. The important bit is to ensure that this frame is large enough such that the child's pager never needs to refill either of its slabs while executing in the parent.

2. Performing all the necessary child vspace mappings (as required by the elf loading process and the argument frame etc).

3. Mapping the pager frame into the child vspace (now that we know that the elf load doesn't need to perform any more mappings at fixed locations anymore).

4. Adjusting all the pointers in the child pager by the offset given by (location in child vspace - location in parent vspace) of the pager frame.

5. Finally copying the `paging_struct` itself into the pager frame too and passing the child's virtual address to the pager frame in the `spawn_domain_params` struct.

As you can probably tell, step 4 is not entirely trivial to implement and caused in its merry share of pagefaults. Currently, the slab allocators themselves are not passed along but instead re-initialized in the child - this has the drawback that we not only lose all of the free slabs the allocators still had when passing the structure to the child but we also cannot reuse the existing ones if the child decides to free any of its initial mappings. The reason why we aren't adjusting

the slab allocator's pointers is because it is particularly cumbersome - but it is definitely future work to keep in mind.

3.4 How multithread support solved the refill issues

Not sure if this should go here or into the multithread chapter. Probably here is better (actually, above where the previous refill issues are described)

3.5 Page fault handling

The goal is to have a `malloc` function that only returns us a virtual address not backed by any physical address. When we access that memory address, we trigger page fault and while handling page faults, those reserved virtual addresses are eventually backed up by real memory.

When `malloc` is called, the pager is asked to save a range of virtual addresses and mark the range as "reserved". Since such virtual addresses are not actually registered in MMU, when we access them, a page fault will happen. When a page fault happens, the page fault handler quests the pager whether the page fault address is reserved. If yes, the handler will request a frame of size `BASE_PAGE_SIZE` from the mem server (via a RPC call). The received frame will be paged in to back the page-fault address. The instruction that triggers the page fault will be re-executed but this time the MMU will approve the memory access.

Our page fault handler can also be used to achieve dynamic the stack size increment, which is a bonus task. During the creation of a thread, the thread stack is half-allocated (i.e, backed) and half-reserved. When the allocated stack space runs out, a page faults will happen and the handler will transparently enlarge the stack size by backing reserved regions with real memory. Note that we do not support unlimited stack size increase because we think a hard limit on stack size should always be imposed to prevent the stack from growing rampantly. Our approach, also comes with a downside that we are reserving too many virtual addresses since most threads will not run out of stacks. In a system with 2 GB of memory and a 64-bit address space, we would argue that it should not be a big issue.

Sidenote on `malloc`. We have seen use cases of `malloc` in the code before the memory server is set up. Page faults from these `malloc` will fail because no memory can be obtained without the memory server. Therefore, we stick to using the "no pagefault" `malloc` (backed by static memory) until the memory server is set up. Once a domain can contact the memory server to request memory, it will switch to the "page fault" `malloc`. As the usage of "no pagefault" `malloc` is during domain initialization and thread creation, the amount of memory "malloc-ed" is very predicable, if not fixed. Therefore, we argue that there is no risk of using "no pagefault" `malloc` during initialization.

Chapter 4

Multi-threads

4.1 Challenge of multi-thread support

4.1.1 Data structures

The introduction of multiple threads poses a great challenge on our data structure because they are not designed to be thread-safe at the beginning. For example, concurrent access to a client rpc call may overwrite a thread's message. We need to provide thread-safety for the following data structure/library.

1. Memory manager. Concurrent access may lead to the free region tracking data structure to be inconsistent.
2. Pager. Likewise, the pager's book-keeping data structure for free virtual addresses should be protected.
3. Client RPC calls. If not protected, the receive slot allocated by one thread may be used by another thread. Similar for the request message.

In addition to the above data structures, we also add thread-safety to the slab allocator and slot allocator.

4.1.2 Refilling

The refilling hazard caused by re-entrancy has been discussed in 3.2. The refilling issue can only be exacerbated and become more convoluted with the presence of multiple threads because now not only different control flow but also multiple thread can cause the trouble.

4.2 Overview of our locking approach

4.2.1 Why not just a big, global lock?

Trivially, the aforementioned issues can be solved by having a global lock that oversees all critical sections. With only one lock, we can also avoid deadlocks. However, it is obviously a sub-optimal solution – many independent operations now need to be strictly serialized. This severely slows down our system.

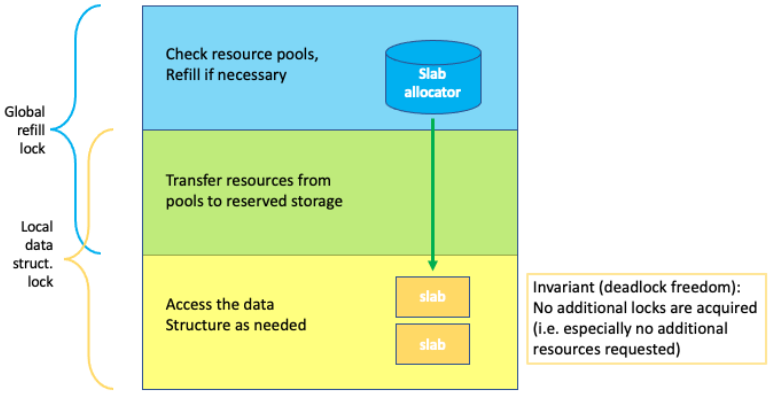


Figure 4.1: 2-type lock approach that provides the fine grained locking and prevents deadlocks.

To provide more fine-grained locking, we proposed and implemented a 2-type lock idea. The two types of locks we have are

1. A domain-global refill lock. By acquiring this lock, a thread can refill the resources it needs. More importantly, it should save the resources to a thread-local storage so that a subsequent thread acquiring this lock will not embezzle the resources.
2. A local access lock per data structure. With this lock acquired, a thread can safely manipulate the corresponding data structure or rpc call.

With the 2-type lock approach, accessing a data structure would look as follows (A pictorial illustration is shown in Fig.4.1.

1. Acquire the global refill lock
2. Check if the slab and slot allocators etc have sufficient resources to complete the access without falling below their watermarks.
   (a) If yes, go to step 3.
   (b) If no, perform refills as needed, without releasing the global refill lock.
3. Acquire the local access lock. While both locks are held, transfer resources that we refilled in the previous step into the "reserved storage" area of the data structure. Finally, release the global refill lock.
4. Perform the data structure access (critical section), release the local access lock.

4.2.2 Locking invariant

The 2-type locking approach provably prevent deadlocks. That is because this approach provides a very important invariant to our system – *if a local access lock but not the global refill lock is held, the data structure access is guaranteed to succeed without needing to acquire additional resources.*

With this invariant, we claim that no deadlock arisen from these two types of locks is possible. We sketch the proof as following. Deadlocks can only happen upon lock acquisition. Suppose thread A wants to acquire the global refilling lock while B is holding it right now. Because of the lock ordering, thread A cannot hold any local access lock needed by B. This means thread B will successfully acquire the local access lock it needs and then unlock the global refilling lock. In another scenario where thread A tries to acquire a local access lock held by B, according to the invariant, B will successfully leave the critical section without obtaining more locks. In summary, whenever a thread attempts to acquire a global refilling lock or local access lock, as long as the invariant holds, it will also succeeds. Therefore, no deadlock is possible.

4.3 Data structure adjustments

In description of the 2-type locking idea, we emphasize that a thread needs to save the resources it needs to a special reserve area while holding the global lock. We call this "resource refilling is not resource acquisition". This is necessary for keeping the invariant described in 4.2.2. Imagine thread A, while holding the global refilling lock, refills the slot allocator but does not allocate the slots. Then it unlocks and tries to page in a physical frame. While it allocates slots for the VSpace, another thread B steal all the slots that thread A refills for other purposes. Thread A will end up with insufficient resources even though it is holding the pager local access lock (break the invariant).

For this reason, for each data structure, we added a reserve resource pool for "hoarding" the resources it may need when holding the local access lock. Because we do not know the exact amount of resources (which is determined by run time dynamically), we simply save the maximum possible needed. After the adjustment, the pager, as an example, looks like Fig.4.2.

4.4 The problem of unbacked stacks!

While testing our 2-type lock approach, we ran into deadlocks. The investigation shows that the stack space of a thread is also allocated by `malloc` in the provided code and therefore page faults are needed to back up the stack with real memory while the thread is running. If the stack page fault happens during the critical section, our invariant no longer holds, because the thread has not acquired enough resources to handle the sudden page fault. (The page fault handling requires a RPC call to the memory server and the pager to allocate slots in VSpace.)

We realize that this issue can lead to other hazards if the page fault happens at an unfavorable moment. For example, if page fault happens while the dispatcher is disabled (for example acquiring a spin lock), we will run into an unrecoverable `page fault while disabled` error.

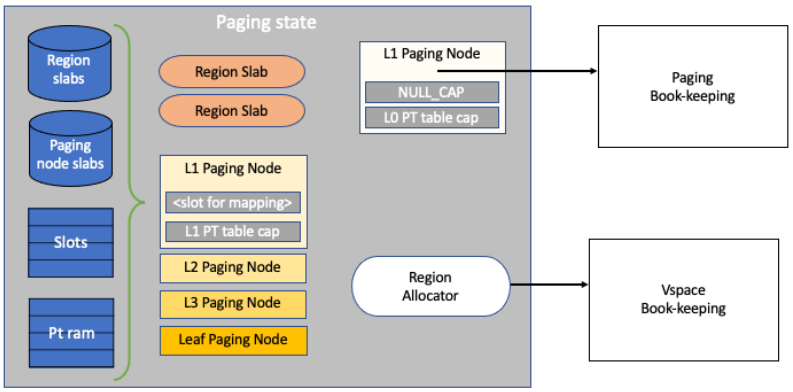


Figure 4.2: The pager data structure after adding in reserve resource pool (on the left).

More details on this and our solution is discussed in the next section.

4.5 Page fault handling revisited

Now that our system is multi-threaded, we also need to enrich the page fault handler to solve the following issues.

1. Multiple threads can page fault at the same address (or addresses that belong to the same page). We have to prevent multiple threads from requesting frames for the same virtual region.

2. Since the handler may make a RPC call to the memory server, it also needs to acquire locks. Together with the the page handler's own lock, we have to be careful of deadlocks.

We will discuss these two issues in more details.

To solve the first issue, we have to serialize the access to the page fault handler. We introduced a page fault handler lock that only allows one thread to execute this exception handling at any time. Suppose thread A and thread B all page fault at the same address and page A executes the page fault handler first. When thread B executes the handler, it will first check if the page fault address is "reserved". Since it has been backed with real memory by thread A, the pager will tell thread B that the address is already "used". Then thread B immediately returns from the handler knowing that the page fault has been handled by another thread.

To understand the second issue, let's consider a concrete scenario of deadlock.

1. Thread A acquires the memory server rpc lock

2. Thread B page faults and acquires the page fault handler lock

3. Thread A later also page faults and waits for the page fault handler lock held by B

4. Thread B, attempting to back the memory, waits for the memory server rpc lock held by A. Deadlock!

To mitigate this, we design a pre-condition for entering the handler. The condition is "the thread holds no rpc locks, global refilling lock, pager lock and slot allocator lock before entering the handler". In the above example, thread A should not page fault when holding a memory server rpc lock. Alternatively, if thread A holds a global refilling lock while entering the handler, thread B cannot progress either because it is the first lock to acquire for making the rpc call to the memory server. The pre-condition is in fact equivalent to saying that for the critical section guarded by those locks, no page fault should occur. This implies we cannot use `malloc` in such critical sections.

Chapter 5

Processes

5.1 Introduction

In this chapter, we will discuss the process of creating a new dispatcher, thus creating new processes. We will first briefly describe the steps done to spawn a process and then move to our decisions and own ideas.

5.2 Our implementation

As clearly pointed out in the book, we can summarize our work in spawning processes in the following steps:

- Load the binary
- Load the command line
- Find and map the ELF file
- Create and set up the C- and V- spaces
- Parse the ELF file
- Initialise the spawn domain and dispatcher
- Passing the paging state to the child

We are now going through the process step-by-step.

5.2.1 Load the binary and cmd line

The first thing to do is load the binary from the memory and map it into the parent v-space. The parent specifies the name of the binary to be loaded and the `multiboot` functions, in conjunction with the `bootinfo` data structure, will find and load the binary. The very next step is to load the cmd line used to pass the arguments for the process and consequently create all these arguments by parsing the loaded cmd line. To fulfil this task we were provided with handy functions in the `multiboot` file such as `multiboot_find_module` and `multiboot_module_opts`.

Since our final system will have a filesystem, loading the binary and parsing the ELF (which will be described in the very next section) could be done from the sd card. More information about the operations involved in the process can be found in the Section10.4.

5.2.2 Find and map ELF

As already mentioned, the ELF file needs to be mapped in the v-space of the parent. For this task, we can use the paging mechanism already described in the previous chapter. In the end, it is worth checking that the first 4 bytes of the mapped binary are the bytes 0x7f 'E' 'L' 'F'. This will confirm that we succeeded in loading and mapping the ELF.

5.2.3 C-Space and V-Space creation

The process of creating the C-Space can be divided into 3 important tasks:

- Create the required C-Space structure
- Create the dispatcher
- Copy the relevant capabilities inside the new C-Space

The first two aspects are straightforward and well described in the book indeed, we neither encountered difficulties nor we had to make important design choices. The third one required us some modification to support our design choices.

In particular, we had to add the newly created well-known capabilities for our initial domains, namely MM_SERVER, SERIAL_SERVER, and SPAWN_SERVER. Later in the development of the OS, we had to add some more well-known capabilities. In particular, we had to support one new well-known domain (at least on core 0) which is the FS_SERVER. The new domain required their own well-known capability indeed all the processes should be able to contact them. Moreover, as it will be described later, to support the shell, filesystem and network we need to pass to those specific domains also the cap_dev and the cap_irq which are needed to have access to devices registers and to set up interrupts respectively. All these pieces of information are stored in the spawninfo struct.

Moving on to the setup of the v-space, the most relevant operation is the creation of the root page table for the child. The new capability needs to be copied into the child's c-space and can also be used to initialise the paging state of the child.

5.2.4 Parsing the ELF

To parse the ELF file we need to do two things:

- setting a callback function to be used when the elf will be loaded;
- call the already provided `elf64_find_section_header_name` function to get the section header of the binary

The second point does not require more explanation while for the first one we had to pay attention to some details. Indeed, this function caused bugs in our first attempt at spawning a process. The function is required to do 3 simple operations:

- allocate memory using `frame_alloc`
- map the new frame in the child v-space
- map the new frame in the parent v-space

The double mapping is required since the child will need to read from that frame the information related to the ELF while the parent needs a way to write that information. The subtlety is that the base address and size given might not be a perfect fit in memory so adjustments to both of them are required to make the address a multiple of `BASE_PAGE_SIZE`.

5.2.5 The last steps

To conclude the spawning process we had to fulfil the following tasks:

- Set up the dispatcher
- Set up the arguments
- Start the process

As the one before, even these tasks were straightforward and well documented in the book. Since the space for design choices is limited we will only talk about coping pager information to the foreign c-space domain and then move on to the process management. In fact, to conclude the spawning processes we had to copy over all the paging information that the child should already know at startup. In particular, we had to add to the paging state of the child all the information used during the v-space setup. To do that we defined a new function whose aim is to copy all the capabilities that refer to tables, frames and mapping into the child root page slot.

5.2.6 Processes management

Process management is achieved using a linked list where all the spawned processes are added at spawn time and then removed once the exit signal arrives. We will talk more about the exit signal in the chapter dedicated to the shell, while here we will focus on the actual mechanism of maintaining the data structure.

The linked list is composed of spawinfo structs which are created while spawning the process. Inside this struct, we decided to also store the PID of the process. This informations is not unique among the cores and is an incremental value maintained by the domain in charge of the spawning process: the spawn server. We will talk about it right after this section. Concluding on the linked list of processes, we designed a lock system to ensure that the list is thread-safe, thus we had serialized the access to the list both for writing and for reading.

5.3 Spawn Server

To follow the spirit of the system, we decided to delegate the spawning task to a different domain, outside init. This domain, called SPAWN SERVER, is present on every core and, once started can be contacted using RPC calls and be asked to pursue one of the following tasks:

- Spawn a new process
- Acknowledge the end of a process
- Kill a process
- Return the list of PIDs
- Return the name of a process based on its PID
- Check process existence

The first task and the ones regarding PIDs have already been explained. Moving onto the process ending phase, we changed our approach throughout the course. Indeed, at first, we simply developed what was suggested in the book so we added a new operation to the dispatcher's capability. This approach had a major drawback, indeed, if the process was not in the run queue due to a deferred call such as `barrelfish_usleep()` then the syscall won't be able to kill the process. The process will be readded to the run queue as soon as the function that has removed it ends. While working on the last individual milestones the strategy has changed. More information will follow, especially in the "Capability revisited" project (see Chapter 9) but we will anticipate something here to give a better understanding of the spawn server abilities. First of all, we implemented the libc_exit function which is in charge of notifying the SPAWN_SERVER when a domain is exiting. This will trigger an update in the list of active processes and, indirectly, on the number of active processes. Then, thanks to the ability to revoke capabilities we are able to revoke the dispatcher capability and actually destroy it while reclaiming its memory. The same approach can be used to support the killing of a process. Indeed the spawn server will act as if the process notified its exit even though the request received is different and, more importantly, usually comes from another domain.

5.3.1 Spawn server struggles

We had to admit that we struggled in the first attempt of setting up the spawn server in a separate domain. At first, we did not realise that we did not pass all the ingredients needed. Indeed we discovered the hard way what we needed to pass and map. In particular, we forgot to pass `cap_mmstrings` and to map the bootinfo inside the child paging state. However, even if we had to scratch our heads for a while, if we look back at the process now everything makes sense. In the end, we are satisfied with the choice of pursuing the domain division for the spawn server.

5.4 Spawning order

Since we have quite a few well-known domains, it is worth talking about the spawning order and how we ensure the correct functionality of the system. Briefly, the spawning order is:

- Init
- Mem Server
- Serial Server
- Filesystem Server
- Network Server
- Spawn Server

The order is somehow logical. In particular, we decided to use init to spawn all the well-known servers and only move to the spawn server after it. This decision was made to be sure that all the endpoints for the well-known services were already been established and can be used by the spawn server while spawning all the user processes. Moving on to the rest of the orders the key points to decide on were:

- Avoid dependencies
- Outsource as soon as possible

Since all the processes need memory we decided that the memory server would have been the first one to be spawned. After the initial setup and registration with init, which will be fully described in the chapter about RPC, init itself will send to the memory server the capabilities for the free memory regions to be used in the memory manager. It is important to note that all the domains need memory to start and can only have it from the memory server after they were able to initialize a connection with it. Thus, to cover that gap, we use morecore_alloc_nopagefault until we manage to have a connection with the memory server.

After that, the second one to be spawned is the serial server. Indeed this does not require talking with the other subsequent well-known domains but will be used by them all to print and read from the serial interface (in userspace).

The file system server and the network one does not have a specif order in our system but it felt more logical to set up the file system before the network, thus we persued that way.

Last but not least, we spawned the spawn server. As mentioned before, all the well-known endpoints are now available and can be inserted in the c-space of the spawn server which can later use them while spawning other processes.

This concludes the system startup (at least for now, in the next chapter the multicore support will extend this startup process) which is now ready to be used. We usually used this point to spawn the tests used to demonstrate all the milestones up until now where we spawn the shell.

5.5 Performance Analysis

For the performance analysis of the spawning mechanism, we instrumented the code to measure the execution times of the different sections mentioned in this chapter. The measurements were conducted on the final system, meaning that two cores are running (see Chapter 6) and the spawning is performed through the built-in "oncore" command of the shell (see Chapter 8).

For each running core, a process was spawned ten times. The figures in this section show the average time of each task over these ten spawning calls.

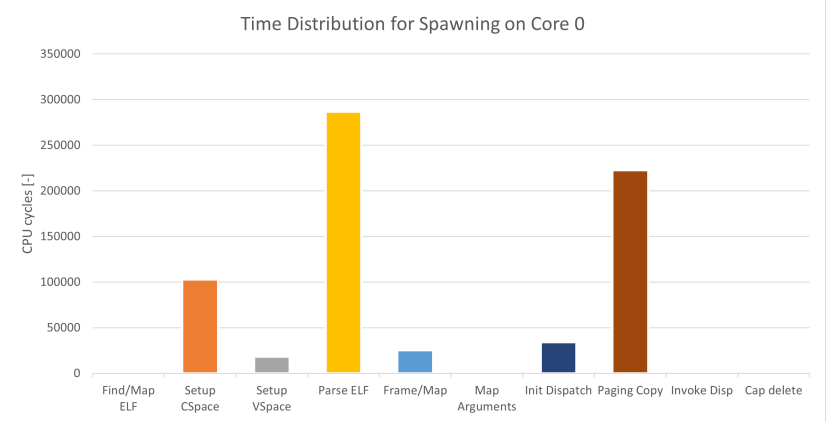


Figure 5.1: Time distribution when spawning a process on core 0 from the shell. The shell is running on core 0. Most time is spent parsing the ELF file as well as copying the paging state to the spawned process.

Figure 5.1 shows the execution time for each task that needs to be performed in order to spawn a process when spawning on core 0. About 42% of the execution time is spent parsing the ELF file of the to-be-spawned process. Another large task (about 32%) is the copying of the paging state to the child process. The setup time for the child's CSpace is the third largest time contributor with about 15%. These three tasks make up around 90% of the total spawning mechanism.

Figure 5.2 shows the same distribution for spawning on core 1. The same three tasks make up the vast majority of the execution time. However, every task takes less time than on core 0. This difference is shown in Figure 5.3.

5.6 Conclusion

Before moving to the next chapter, we would like to summarize what has been done and which are the possible improvements. Indeed, even if this task was not that difficult at the beginning except for the multiboot part already mentioned, it has a key role in the startup process of the system. In general, we are satisfied with the overall design but our process management system does require adjustments every time we added a new well-known domain which is spawned before the spawn server and not on all the core. Moreover, the data structure

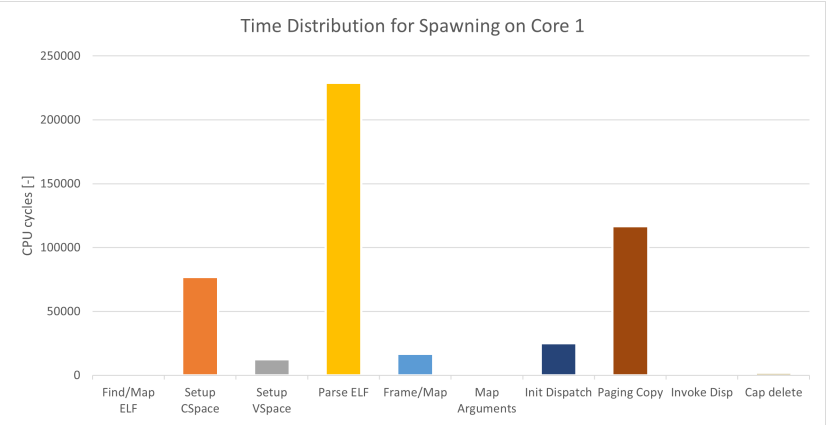


Figure 5.2: Time distribution when spawning a process on core 1 from the shell. The shell is running on core 0. The same three tasks make up the majority of the execution time as for spawning on core 0.

holding the process can be substituted to use a tree so that the search can still be done fast even with a substantial increase of processes.

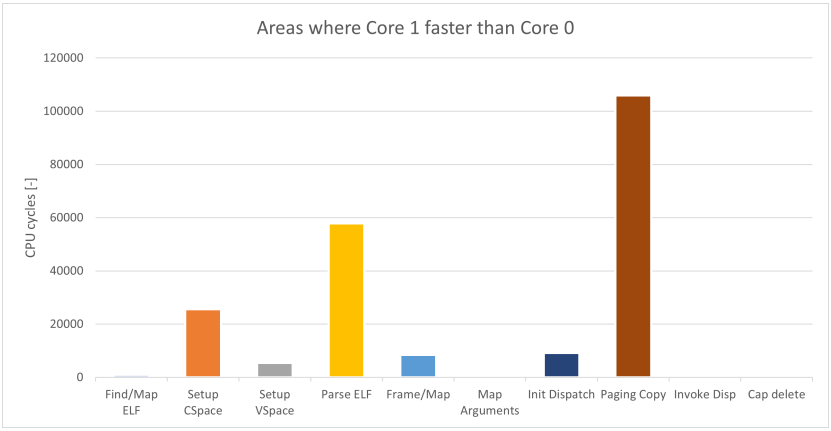


Figure 5.3: Difference between spawning a process from the shell on core 0 and on core 1, grouped by task. The shell is running on core 0 in both cases. Every task is faster when spawning on core 1 compared to spawning on core 0.

Chapter 6

Multi-core Support

6.1 Introduction

Having a multi-core CPU is the most common setting nowadays because it enables true parallelism (rather than context switching on a single core). In Barrelfish, each CPU core runs a CPU driver that serves only the domains on that core. Different CPU drivers do not share any common states by default. This gives us a very symmetric view among the CPU drivers – they are almost identical and work individually. The symmetry makes most our code developed for the bootstrap core reusable.

Previously we have been developing pager, memory allocator, processes and so on for the bootstrap core. This chapter will talk about how we managed to boot the other cores and let such services continue on the those cores. We will also discuss the initialization process of the app cores in which we need to establish the UMP channels, populate memories etc.

Most of the work for this chapter follows the instructions from the textbook. Things we innovated are how we pass the `bootinfo` and how memory are allocated and transferred to other cores. We will detail these two aspects in the following sections.

6.2 Memory division among cores

In this section, we discuss how we divide the memory among different cores. In general, we want to allocate the same amount of memory to each core. During the development of the OS, we had to change our approach to accommodate the "capability revisite" project. However, since for the majority of the time a different method has been used, we decided to present both.

6.2.1 The first approach

When the BSP core is turned on, its `init` domain adds all the free memory regions specified in `bootinfo` to the memory manager. Then it spawns the well-known services (e.g., memory server, spawn server etc). After the memory server is up, `init.0` will send the free regions to the memory server and let it manage them hereafter. Notice the free regions that `init` receives are different

from what memory server receives because `init` may have used some memory already. The same applies to the scenario when we want to boot other cores. We have to make sure that the BSP core only gives out free memory. To solve this, we ask the memory server to allocate the memory that will be assigned to an APP core because the memory server keeps track of the free regions and used regions. In this way, BSP core will not give out memory in use.

Assume we have 1600 MB of free memory to be assigned to 4 cores, it can happen that the memory server cannot find a continuous free regions of 400 MB. If that happens, we will divide the 400 MB into 2 200MB regions and retry. This division continues to happen until we have a set of regions whose sizes add up to 400 MB.

The free regions will be written into the `bootinfo` sent to the APP core. The APP core will read and initialize from the `bootinfo` the same way that the BSP core does (hence reusing the code).

6.2.2 The second approach

The approach explained above did not sit well with the requirements of the "capability revisited" project. Indeed, the memory capability that every core holds should be root capabilities in order to make the reclamation possible. However, more details on that will be given in the chapter dedicated to that project. Let's now talk about the new approach.

For this approach, we are restricted to two cores only (as required to make the new capability system work). We decided to split the root capabilities right before adding them to the memory manager. In particular, all the root capabilities with a size greater than 2 `BASE_PAGE_SIZE` (which is the minimum amount of memory that can be asked to the `mm_server`) will be retyped in two new root capabilities and then deleted. One of those two new capabilities will be added to the memory manager of the current core while the other one will be stored in a linked list to be used later during the second core booting process.

To conclude, even if our final solution is not implemented to support 4 cores, it is fairly easy to design a data structure that holds at most 3 linked lists. The same applies to the capability of retyping. Thus we consider our solution to still be fairly general and scalable.

6.3 Passing the multiboot info

In the book, we are suggested to send the `bootinfo` and memory to the app cores after the cores are up. We think a better way of doing this is storing these information in the shared memory before booting the second core so that the second core can already get the `bootinfo` and memory it needs once it is up. This benefits us in the following ways.

- The app cores can spawn new domains or allocate memory as soon as it is up. Because the available memory regions are stored in the shared memory, we can forge them and add them to the memory manager.
- We can reuse the most of the code developed for core 0. The BSP core, at its inception, has all necessary information (esp. `bootinfo`) to do the

initialization. By following our method, the APP cores initializes itself as if it is a bootstrap core by using all the functions BSP core uses.

- It save us from the tediousness to develop RPC calls just for passing `bootinfo` or ELF modules.

As for implementation, we uses the URPC frame shared between as BSP core and an APP core to pass the information like `bootinfo`, memory regions, ELF modules etc. The app core in `app_main` will read the information out of it, initialize relevant parts of the system and notify the BSP core that it is ready. The notification uses the the first byte of the URPC buffer as a flag. The BSP waits for the flag to be cleared. After the first core has been acknowledged, the URPC frame will be used as ring buffers for the real URPC.

6.3.1 Signalling core readiness to use UMP

After the BSP core has been signaled the readiness of the APP core, the cores will simultaneously initialize the UMP client, server and forwarding channels. This initialization involves no communication, i.e., no writing to the URPC frame. Instead, each core simply sets the correct pointers for the respective ring buffer. After the channels are established, we can do URPCs bi-directionally. The sequence of actions between the start of the second core and being ready for URPC calls are depicted are summarized in Fig.6.1. We defer the detailed discussion to the URPC to the next chapter.

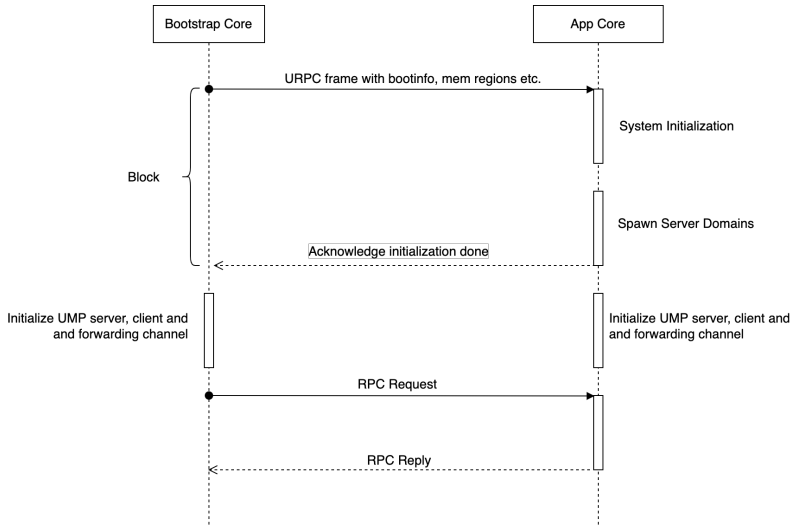


Figure 6.1: The time sequence diagram of how cores prepare before URCP starts to work.

Chapter 7

Remote Procedure Calls

Remote procedure calls (RPC) provide abstraction that allows an application to call remote functions as if they were local. The client application passes in the arguments and the RPC library will send arguments to the server and then return to the application the server's reply.

7.1 Overview

We designed the full architecture of RPC in the chapter of LMP. The stack greatly helped us achieve separate of concerns during development and provides powerful abstraction layers that enables us to implement our idea modularly and productively. The stack is illustrated in the Fig.7.1.

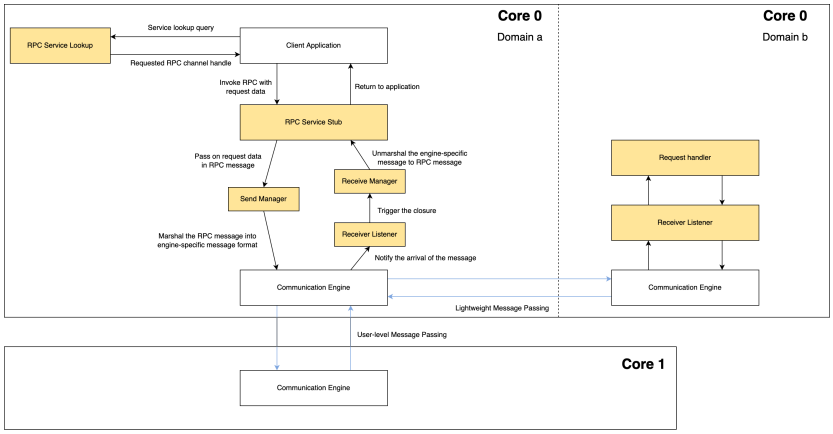


Figure 7.1: RPC Architecture

On the top left (Core 0 domain a), we show the RPC stack for a client. The boxes in orange are provided by the RPC library. We introduce them one by one.

1. RPC service lookup. This module returns the RPC channel handle to the client for the subsequent RPC call. If it is simply a getter function,

it is not actually "look-up'. In fact, we are able to allow the user only declare which core and domain he/she wants to reach, the look-up service will automatically return the most suitable channels to the user because it maintains a collection of all available channels (like a route table). Since sometimes messages can be routed more than two ways between two domains, the look-up service is able to return the best one based on the message size, whether a capability is going to be sent or not etc.

2. RPC service stub. The stub contains interfaces that a user will use to make RPC calls. The stub provides the abstraction that making a request to a remote service is like making a local function call. The interface does not directly interact with the communication engines, i.e., LMP and UMP. Instead, it parses the user request into a `rpc_message`. This message format is agnostic to the communication engine.

3. Send manager. This module marshals the `rpc_message` prepared by the service stub into an engine-compliant format. For example, for LMP channels, it would be 4 words and for UMP channels, it would be a series of cache lines interleaved with flags. Moreover, send manager provides different ways a message (whether it be a request or a reply) can be send. For example, the send can immediately return once the message is sent, or wait until the reply is received.

4. Receive manager. Receive manager does the opposite of the send manager. It un-marshals the raw message directly returned from the channel and parses it into the `rpc_message`. It also provides different ways of receiving. For example, a single thread can receive for other threads and multiplex the message to the corresponding thread.

5. Receiver listener. This light-weight module mainly listen to the channel and when the message has arrived, it will react by either call the correct handler.

6. Communication engine. This is the module that actually implements how the messages are actually sent or received. For LMP, the code is provided and for UMP, we implemented it ourselves. The communication engine is abstracted as an `aos_rpc` channel.

Send manager, receive manager and the receiver listener constitute the core of our RPC library. We call them altogether *message manager* as it translates between raw message and `rpc_message` and interacts with the channels to send and receive messages.

7.2 Naming

All the possible interactions that can happen between two domains are specified in the aos_rpc file. This file, which already contained some predefined RPC calls, has been deeply expanded to support all the OS features. This only represents the client's perspective but for the communication to succeed it is required that another domain acts as a server and allows queries for that particular message. Indeed, RPC calls are wrappers around an exchange of messages

between two processes acting like client and server respectively. As already said, these communications happen over a channel which can be of different types. Those channels will be meticulously explained in the following section, but to have a high-level overview all the interactions can be classified into four types:

- Direct connection between processes on the same core
- Direct connection between processes on different cores
- Forwarded connection between processes on the same core
- Forwarded connection between processes on different cores

7.2.1 RPC client interface

To make those RPC calls as general as possible, all the complexity of channels is hidden in a layer below the one used for the communication itself. However, we still had to set up the correct message and handle all the possible responses for every case. Another option that we decided to support, is to create RPC calls where the client does not need to set up the channel but just specifies the domain and the core id where the request should be sent. This second option has the scope to find the best channel already present in the system to perform the action required by the user hiding this complexity from the client itself.

All this interface is translated into code by a set of tree functions for every RPC call. As already said the user can either interact with the normal call or the "target" one which is the one that does not require the specification of a channel. The third function is a shared helper which will take a more complex struct, the `chan_msg_arg`, which will be explained later and performs the communication. The return value or error is then propagated back to the client. Indeed, from the perspective of the client, all the RPC calls are blocking.

7.3 Binding and Connecting

7.3.1 Infrequent communication

Our system does not strictly require the creation of direct channels. Thus, in case two processes need to perform infrequent communication between each other, they can rely on the forwarding channel infrastructure to have their messages delivered. In particular, the process acting as a client will get the channel to the init server present on its core. Comparing the channel type with the header of the request, init will continue the forwarding process sending the message to the init present on the destination core. That init, will then forward the message to the correct domain which can serve the request and reply. The reply will follow the same route. More information on the actual forwarding mechanism will be presented in the Section 7.6. However, it is important to specify that not all the messages can be sent over the forwarding channels, indeed, there is a limitation on the payload size (which is 32 bytes). In that case, a direct channel will be silently established and used instead of the forwarding one.

7.4 RPC Message abstraction

The RPC message is an important abstraction that bridges the RPC stub and the actual message passing mechanism. As there are multiple possible ways to deliver a message between two dispatchers, RPC should be able to translate user's request into engine-specific message format. RPC message plays an important role in letting the RPC stub easily understand the semantic of the message and letting the message manager marshal and unmarshal the message.

Every RPC message is comprised of the following elements, each of which will be discussed in more detail in the following subsections:

- A representation of a capability
- A header
- A payload

At first glance, it might seem more sensible to embed the representation of the optional capability that accompanies that message into the payload. The reason why we are not doing this is because the low-level lmp send and receive functionality takes capabilities and message contents (represented as words) separately.

Of course, this makes a lot of sense because the lmp mechanism has to be made aware of which bits should be interpreted as capability references to make the respective capability available in the receiver's CSpace, and which should be transmitted as they are.

Thus, to make the task of the message manager as simple as possible and prevent it form needing to identify the message type and pull out the capability that is possibly included in it on the sending and "shove it in" again on the receiving side, the capability description is a distinct.

Additionally, this design decision allows for the more verbose cross-core capability description described below without it using up any of the precious 32 bytes that a single lmp transfer can carry.

7.4.1 Capability representation

Messages can optionally transfer capabilities. To accommodate this transfer, regardless of the underlying transfer mechanism, we need to be able to flexibly between two different representations:

1. A capref in the current CSpace for lmp transfers
2. A more verbose description of a capability for the transfer of capabilities and the synchronization on capability operations between cores. To make the monitor's job as easy as possible, this description has room for all the possible information that it might need to include depending on the operation, namely:
    - **The capability struct**
    - **The owner of the capability** This is used when transmitting capabilities between cores such that the receiver can pass the correct arguments to `monitor_cap_create`

- an ownership transfer indication For transmissions that also transfer the ownership of the cap to the receiver; if set, the receiver additionally needs to adjust the owner of any existing copies it has to avoid inconsistencies (and the resulting assertion failure in cap create)
- **A Remote relations bitstring** Also for ownership transfers: the receiver needs to be made aware of whether the capability had any local descendants on the sending core to correctly set the remote descendant flag (since it's not possible for a core to faithfully track the remote descendants bit on a remotely owned capability)

If no capability is supposed to be sent across, we indicate this by adjusting the representation to symbolize an empty capability: For caprefs, this is the `NULL_CAP` while for the verbose description we set the type of the capability struct to `ObjType_Null`.

7.4.2 The Header

The header of the message is a late addition: it only became necessary with the introduction of cross-core communication, which introduced the additional complexity that a message may need to traverse multiple domains to reach its destination.

For example, if two non-init domains on different cores wish to communicate, they can do so over a direct channel in our design. However, the setup of this channel still requires the involvement of their two inits, because the required shared UMP frame cannot be conjured out of thin air but needs monitor support.

The main purpose of the header is to indicate to any such intermediate domain what the ultimate destination of the message is to allow it to correctly forward it. More concretely, it indicates:

- The destination core
- The destination domain
- The source core
- The source domain
- A message sequence number

The usage of the sequence number is discussed in more detail in the message manager section below. We also include the information about the source domain to correctly forward the resulting reply back to the sender.

7.4.3 The payload

Different RPC calls require different information to be passed between client and server. The structure of the payload is therefore designed in a way that provides a common denominator for all these different types of RPC messages, while also allowing them to carry individual information.

As a result, it is comprised of a big (!) union of **all** message types used in the client-server RPC interactions, all of which feature have the unique identifier of their message type as their first field, which (according to the C standard) allows

that identifier to be correctly accessed and interpreted without the message type being known.

As a result, the handling of RPC messages on the server-side can be a "simple" case statement that conditions on the message type to perform the appropriate actions and construct the appropriate reply.

Additionally, the payload structure must provide support for the message manager to correctly identify the length of a given message such that it can correctly marshal and demarshal the messages onto the communication channels.

For fixed length messages, this problem is easily solved: we simply provide a function that maps the messages to the size of the structure used to present their type.

For variable length messages, things are a bit more complex: however, we would first like to draw your attention to the fact that all the RPC calls that we are required to support, as well as all the additional calls that we added for our individual projects only have at most one element that exhibits variable length and this element is always an array type (because if the variance stemmed from a union of types, we could decompose that message type into different types for each union member).

Thus, the following solution presented itself to us: we place this variable length array (if any) at the end of the message type struct as a flexible array member and then add the length of this array (in bytes!) as a second common field that shared by all variable length types.

We then provide functions that indicate if a message type has variable length and that provide the offset of the flexible array in the struct, which the message manager can use to calculate the size of any message that it encounters.

Figure 7.2 summarizes the layout of the payload structures.

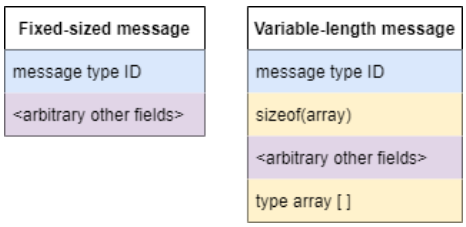


Figure 7.2: RPC message payload structure for variable and fixed-size

7.5 Routing and channel selection

Init knows all, so if in doubt, forward to init.

7.6 The RPC Channel abstraction

Our design represents both client and server-side channels (i.e. the channels used by clients to send to / receive from a server as well as the channels used by the server to receive and reply to a client) by `aos_rpc` structs.

This choice was made to allow for very uniform handling of send and receive patterns - after all, at the lowest level sending and receiving over lmp or ump

mechanisms is symmetric for both sides. The aspects that differ are the actions that happen between or wrap the this sending and receiving.

As a result, the nature of the represented channel does not only depend on the underlying communication mechanism (lmp or ump) but also on the *purpose* for which the channel is used.

Over the course of various milestones, we continually extended and reworked our channel abstractions (as well as the functionality provided by the message manager) in order to arrive at our final solution. In the next subsection, we wish to describe our intermediate design iterations as well as our motivation for taking the next step forward.

7.6.1 At first, there was LMP

In the very first milestone that asked us to support RPC, we laid the entire foundation for the RPC message abstraction - the only changes that followed were adding in the header and the capability descriptions for the monitor, which were comparatively effortless.

In contrast to this immediate success, our channel abstractions at that point were, as we unfortunately discovered later, not entirely future proof. The main issue was that we coupled channels and messages that are being send and received on them: instead of passing channel and message to the message manager as separate, explicit arguments, we only passed the channel, which contained a pointer to the rpc message.

The manager would then, depending on whether a receive or a send should be performed, either send the pointed-to-message or allocate a fresh message and set the channel's pointer to it. This design choice has the immediate consequence that if a particular client thread performs an rpc call on the channel, it needs to block the channel itself (i.e. acquire a lock on the channel) until the call is handled. Otherwise, different threads may hot-swap the message a different thread was trying to send or receive with their own to-be-sent message, causing a lot of confusion.

While we had been aware of this implication while designing the first channels abstractions, we nevertheless went along with it because of three reasons:

- The client blocking on the channel sounds like it results in abysmal performance. However, with only the lmp case to consider, this does not actually matter: different domains on the same core only run concurrently, NOT in parallel. Therefore, multiplexing an lmp channel does not yield any real benefits - it could reduce the amount of context switches required, but in practice it does not because the client's send flags immediately yield to the server anyway.

- There is an important self-paging issue that comes into play, too: the rpc call to the memory server may be invoked when page faults are being handled (because we need to request ram to back the region in question). As a result, the mem server's RPC call must be *malloc-free* because it's not possible to handle nested page faults.

  The easiest way to accomplish this is indeed to have a message pointer in the channel itself: this pointer points to a pre-allocated message (that we ensure is completely backed by RAM before the channel starts operating).

Clients then populate this message by writing in the data they desire to be sent, while the message manager copies the response data received on the channel back in.

(Note that to fulfill the locking invariant described in the multi-threading section, the receive slot the receive endpoint may need to obtain the RAM cap accompanying the mem server's answer needs to be pre-allocated, too!)

- There is no (easy) way to multiplex the frame that we use to pass large lmp messages (see the rpc message manager section below for a detailed description of this mechanism). Thus, such large messages would need to be queued separately if the channel were multiplexed, which would result in a lot of complexity

So this "incorporate message" into the channel does make a lot of sense - but ONLY for *direct* lmp client-side channels. For different channel types and purposes, this solution is too restrictive as we will see in the next section.

7.6.2 Then, there came UMP

Of course, with UMP in the picture, the design considerations for a flexible and performant RPC framework change. This is because of two reasons:

- There is now true parallelism in the system. Thus, being able to multiplex client channels, i.e. batch multiple client requests on the same ump channel, is lucrative. This is especially the case because each core is shared by multiple domains and so the number of domain context switches can be reduced if we can process more work on the current domain due to multiple pending requests (and thus use the allotted time slice fully). Therefore, having client threads block on the channels is not the most ideal design anymore.
- Secondly, as already mentioned above, some operations such as direct ump channel establishment or sending capabilities need to take the longer route over the two init processes. This has the following consequences:
  - The forwarding role of init does not match either of the existing client and server purposes well - thus, there ideally should be a dedicated channel type with dedicated send and receive mechanisms for this case.
  - The lmp channel that is used by clients to forward over init should ideally be multiplexed between client threads, too: different such forwarded requests may have different destination cores, and thus have the chance to be handled in parallel!

A temporary blocking solution

Of course, the most interesting solution to implement and design is one that is as asynchronous as possible - one would hope that it is also more performant than a blocking solution, but whether or not that ends up being the case also

depends on the overhead introduced by any additional data structures that the decoupling may require.

However, before plunging into the asynchronous hell, we decided to first implement a very simple blocking solution: one, because it was a very low-hanging fruit and could serve as a fallback if the non-blocking solution did not work out, two because it would allow us to compare their performance.

In this solution, clients still block on their channels as before and init performs its forwarding functionality by pretending to be the original client of the request (and therefore blocking on the client channel it has with the next hop along the path until the reply returns).

Since this was a quite simple and temporary solution, there is not much more to say about it, except that one has to be devilishly careful in ensuring that the inits do not deadlock if multiple client domains decide to issue simultaneous cross-core requests:. For instance, what does not work is having only single request handling thread in the init domain: if two domains on cores 0 and 1 decide to perform an rpc call to a destination on the other core, the single request handling thread in both inits could block until the reply to the request originating from its core arrives before handling the request forwarded by the other init.

7.6.3 Take 2: the final, non-blocking solution

For the discussion to make sense, we first define what we mean by *blocking* and *non-blocking* with respect to clients and channels a bit more explicitly:

- **blocking client** A "true" client that used our rpc interface to launch a request. It blocks until the reply arrives, either on the channel itself (in case of a blocking channel) or on a specialized data structure (which we explain in detail in the message manager section)
- **non-blocking (or pseudo) client** A "forwarding server as client" that only forwards the request / reply on behalf of another entity. As a result, it does not block to wait for the reply - we refer to this as **fire and forget forwarding**.
- **blocking channel** A channel where the client thread blocks the channel while it is waiting for the reply. As a result, it does not allow multiplexing.
- **multiplexed (i.e. non-blocking) channel** A channel where blocking clients (i.e. "true clients", if any) do not block on the channel itself but instead on some other data structure (pending_rpcs).
- **forwarding channel** A channel that represents a single "hop" on a "multi-hop" (forwarded) communication path. Is necessarily multiplexed because we use fire and forget forwarding.

Our final, as asynchronous as we could "dream it" solution addresses the design considerations that emerge with UMP (which we discussed in section 7.6.2) as follows:

- UMP and forwarding client channels are **multiplexed**.

- the forwarding performed by the inits is “fire and forget”, i.e. init acts as a **non-blocking client**.

To realise this, we have designed a multitude of different channel types, which are depicted in figure 7.3:

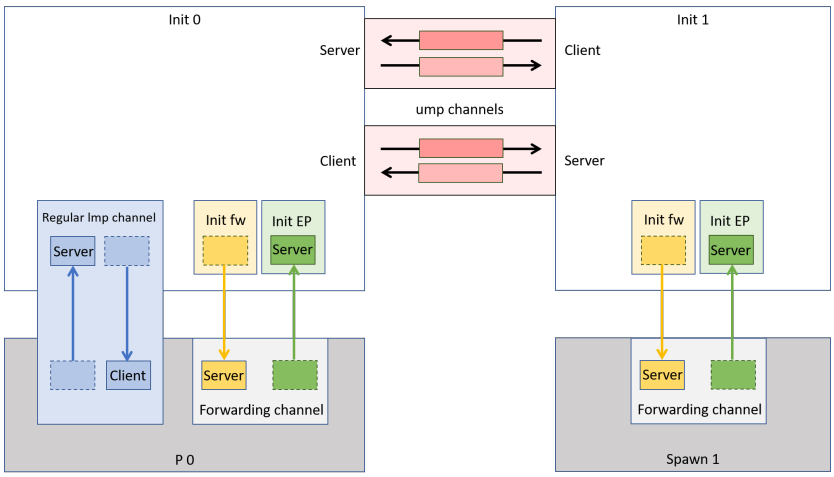


Figure 7.3: Overview of the different channels

In the following, we will take a closer look at each one of these types - their specialized message handling is determined by function pointers inside of the aos_rpc struct, which point to appropriate utility functions implemented by the message manager.

lmp channels

The figure depicts lmp channel endpoints shared between domains as rectangles, whereby the domain that listens on the endpoint has got a copy of it with a continuous outline and a domain that sends on it one with a dotted outline.

regular lmp channels

The blue lmp channel termed *regular lmp channel* is the classic lmp-type channel that we have been supporting since milestone 4: it was created by a connect operation initiated by the client and is thus specifically dedicated to the communication between domains P0 and Init0, as indicated by the closed box drawn around it.

The two channel abstractions by which it is represented in client and server domain are of the following nature:

- **Client view**: for the client, this is a blocking channel: the thread that executes an rpc call on it will acquire its lock, populate the pre-allocated message and call the message manager’s blocking send-receive function, which first sends the request to the channel’s remote endpoint and then blocks on a waitset on which it registered the local endpoint for receiving.

- **Server view**: the server has this channel registered on its lmp-receive waitset (more precisely, the endpoint the server created during the connect operation is registered with the channel as closure argument). A receive event on it calls the message manager's reply-wait function, which *causes* the request to be received, handled and the reply sent back to the channel's remote endpoint. We say *causes* because these individual steps are completely decoupled, as described in the message manager section.

open lmp endpoint channel "constructs"

The green-yellow lmp "construct" is formed by P0's and Init's open endpoints. It cannot be called a channel per-se, but it is rather a combination of channel-views that init and P0 have on these open endpoints, which are used to enable forwarding between them:

- **P0's views**: For P0, the two endpoints represent its unique **forwarding "channel"** - each domain has such a channel in a public structure called *conn_rpc*
  - **Server view** Typically, the server will use this "channel" like a regular lmp server channel described above:

    The only exception are lmp connection requests which will all arrive on this "channel" (although, to be very precise, they need not necessarily originate from init because the open endpoints of well-known domains are passed into the CSpace of newly spawned processes, as mentioned above, so it would be more accurate to say that they arrive on the open endpoint, which just happens to have registered this channel as closure argument).

    These requests are handled in a special manner to complete the binding, i.e. the establishment of the dedicated channel: the server creates a new endpoint and sends it back to the endpoint that accompanied the connect request (instead of the channel's remote endpoint as usual). Like this, client and server are now both aware of the endpoint of their new regular (depicted in blue in the figure) lmp channel.

    Other than these connection requests, arbitrary other requests that were forwarded over init will arrive on this "channel". This is where the beauty in viewing it like a regular lmp server channel lies: the replies to the forwarded requests are automatically forwarded back to init, without the server needing to do anything special.
  - **Client view** As mentioned above, the client usage of the forwarding channel is multiplexed - as a result, clients will call the message manager's multiplexed send-receive function on it. Replies to forwarded requests are "forwarded back" by init to this channel (as indicated by the arrow in the figure).

    Note that as a result, both forwarded replies to client requests of the domain as well as forwarded requests for the domain's server may arrive over the domain's open endpoint - so how do we tell the difference between them? The answer is quite simple - requests always use even sequence numbers in the header, and replies are

constructed by incrementing the requests number by one (and are thus always odd).

- **Init's view** From the point of view of init, the construct of endpoints is actually split into two different, not directly related channels. This of course makes sense because init is special to every other domain, but no domain is special to init, so init's open endpoint will not be associated with any other domain's endpoint.
  - **Init forwarding lmp channel** This is a special type of channel that is created whenever a new domain registers with init. Whenever init receives a request that is needs to forward to that particular domain (which it can tell based on the message header), it does so over this channel. For quick retrieval, these channels are stored in an rb-tree data structure indexed by domain.
  - **Init's "forwarding channel"** Init's forwarding channel, also located in the conn_rpc struct, is exceptional because it doesn't have a remote endpoint - therefore, it is simply called "init EP" in the figure. Without remote endpoint, no sending is possible over this channel and thus it only servers as server listener endpoint that handles requests in the same fashion as other domains.

    Additionally, it is on the receiving end of all messages (both requests and replies) sent over the forwarding channels in other domains - these, init forwards by selecting the appropriate "init forwarding channel" described above

UMP channels

The ump channels are represented by the pinkish boxes in the diagram: Each contains two ring buffers, one for each communication partner to send on and one to receive on - these buffers are represented by boxes with darker shading and arrows to indicate the send-receive direction.

From the perspective of any domain except init, each ump channel serves exactly one purpose: it is either a client channel of a server channel. Note that this immediately implies that all such channels are direct - on a non-init domain, a forwarding path's first hop goes always of its forwarding channel, which is lmp.

- **UMP client channel** as mentioned in our requirements, ump client channels are multiplexed. The client thus call exactly the same multiplexed_send_receive function in the message manager on it as in the case of the forwarding channels (the difference is only in the underlying mechanism, which is ump instead of lmp)
- **UMP server channels** This type, too, is analogous the the regular lmp server channels except for the differing mechanism.

On init, things are a bit more complicated because of our requirement that init's forwarding must be fire and forget, which also applies to in-between-init forwarding, of course.

It turns out that all the different purposes for which communication between inits is needed, a single UMP channel would suffice: the different use cases can

be easily be represented as different aos_rpc channel structs that all point to that single underlying ump channel.

However, for legacy reasons, the two init domains still have two such channels between them (as drawn in the diagram), one for each client-server direction; this is because the mailbox system that is described in the message manager system used to be per-channel but has been unified in the meantime.

To simplify the description, we will describe these different communication channel purposes as if there were only a single channel, because that allows us to talk about *the* ump send and receive buffer without exactly specifying which one (because it doesn't matter anymore):

- **Client and server purpose** Of course, client threads on init might wish to talk to another init because of their own agency and independently of init's forwarding functionality. The heavy decoupling of sending, receiving and handling allows this out of the box, the corresponding channel representations need only set their functions correctly.
- **UMP Init forwarding** Analogous to the lmp init forwarding channel, the ump case is also only used for sending. As a result, the channel representation only points to the send buffer and uses the usual decoupled send functionality provided by the message manager.
- **Monitor purpose** This is the channel view used by the monitor implementation - it is the only channel that allows non-null capability descriptions to be sent over ump. All other send functions will either fail when they detect that the capability is not null (on non init domains) or they well enqueue the to-be-sent message at the monitor for serialization and marshalling (for other init-init ump channel views).

7.7 Message Manager overview

In this section, we will take a closer look at the message manager, which takes as argument the aforementioned `chan_msg_arg` and performs the appropriate send or receive functionality on it, conditioned on the channel type and purpose. We present here the core funcationalities of the message manager.

Provides collection of utilities to support setup, transmission and reception of messages. Conditions on the underlying mechanism (LMP, UMP) to perform correct operation.

Described from high to low-level

7.7.1 Typical rpc operations

Expected reply wait, send receive functionalities that correctly sequence multiple lower level operations.

7.7.2 Specialized wrappers around send and receive

7.7.3 Mailboxing

When a client thread makes an URPC call through the UMP client channel, after the request message is assigned a unique sequence number and then sent

out, the thread registers itself at a pending list and wait on a local condition variable (not shared with any other threads). A mailbox thread upon receiving a reply message will find the target client thread from the pending list based on the sequence number and wake it up.

In this way, we allow multiple client threads to send the requests simultaneously through the same UMP client channel. Each request is distinguished by a unique sequence number. So when the reply comes, the mailbox thread can direct the reply to the correct waiting thread. In essence, the mailbox is a demultiplexer of the reply messages.

Note that we let each client thread wait on a unique condition variable in constrast to on a single condition variable. This prevents the arrival of a reply message from waking up all the waiting threads.

7.7.4 Round-robin polling

Initially, for every UMP client and server channel, we create a thread to listen for replies and requests respectively. We later realize that this idea creates as many threads as the channels, which requires a lot of memory. To remediate this, we creates *only one* thread to poll all UMP channels. This threads will poll on all UMP channels in a round-robin manner. As soon as a channel has a pending message, it will wake up the corresponding client thread if it is a reply (mailboxing), or offload the message to the handler thread if is is a request. If there is no message pending for all channels, it will yield the time slice. It distinguishes the reply and request by the parity of the sequence number – request always has an even seq. number whereas reply always has an odd seq. number.

The round-robin polling will not result in worse performance because the polling thread does not handle the request itself, instead it notifies a handler thread via `event_queue` to process the request. Therefore, the polling thread can continue on polling the next channel. The decoupling of receiving and request handling has another further implication. See the next paragraph.

7.7.5 Decoupling

Another implication of the decoupling is allowing nested RPC calls. Imagine the polling thread (described in 7.7.4 also handles the request. When handling the request, the polling thread also needs to make a rpc call to a remote core (for example, to access the file system server). Because of our mailbox design, the polling thread will sleep while waiting for the reply. That means, the reply to that rpc call cannot be received because the polling thread is sleeping – we have a "deadlock"! Therefore, only with such decoupling can we support nested RPC calls. However, we realize that such decoupling cannot resolve circular rpc calls between two domains on two cores. We treat this as a limitation of our RPC framework.

7.8 Lower level channel operations

7.8.1 LMP message marshalling

The LMP channel of Barrelfish natively supports a transfer of four 64-bit words, meaning that an RPC message sent over a LMP channel can have a maximum of 32 bytes. As the header is a part of every message, not only the UMP messages, 8 bytes are reserved for it. This leaves 24 bytes as the actual payload size of the LMP message. Thus, we carefully crafted all our fixed-size message types to be below this magical threshold.

To make our lifes easier, we wrapped the message headers and payload into a struct and put this complete message struct in a union, together with an array of four 64-bit words. This way, we can easily transform the message into sendable LMP-words.

A problem that emerged with the 24 byte message payload limit is that we cannot directly send dynamically sized messages (such as variable length strings or network data) directly through the LMP channel. This problem is further discussed alongside the solution in the next section.

7.8.2 Large LMP message handling

As mentioned in the previous section, the native LMP channel of Barrelfish only supports a direct transfer of four 64-bit words. Because of this, no large variable length messages can be transferred directly through the native LMP channel. To mitigate this issue, we use a shared buffer (called the "argframe") that is shared between the two endpoints of the channel. As an added benefit, we wanted to make this process as simple as possible to work with from the client's side, so the whole process of setting up the argframe and using it should be completely transparent to the client.

When a new LMP channel is created, the initiating party allocates a frame of size "BASE_PAGE_SIZE" and maps it into its virtual memory. The corresponding capability is then sent to the other endpoint in a special message with the type "SETUP_ARG_FRAME". The other endpoint then also maps the frame into its own virtual memory. This is essentially everything that is needed to set up a shared buffer between the two endpoints.

When sending a message that is larger than 32 bytes, the "rpc_message" struct is copied into the shared buffer in a volatile manner and a special message with the type "LOOK_AT_ARG_FRAME" is sent over the LMP channel.

When the LMP message arrives at the other endpoint, it is first checked if the message type is equal to the special "LOOK_AT_ARG_FRAME" message type. If it is, then the message is copied byte-by-byte in a volatile manner from the shared buffer to the "msg" field of the RPC message struct.

Special care needs to be taken to not overwrite the passed capability reference, hence it is stored locally before copying the message from the shared buffer, and then written back into the message struct afterwards.

7.8.3 UMP send and receive

All the information needed to perform read and write over a UMP channel is stored in two structs: the ump_channel one which holds the higher-level info

and the cache_line struct which is used to directly interact with the cache line.

Writing

Writing is done with a granularity of a cache line. A UMP channel has an index which points to the cache line that should be used to write data and this cache line has a flag which indicates if that cache line can be used for writing. Once the flag goes to 0 we will write all the data inside the buffer of that cache line. Then the indexes are updated and a new write can be done if needed.

Reading

Reading from a UMP is divided into two functions. Although, the concept remains the same. We have to check the flag of the cache line that we should read. Once the check on the flag is done, the second function will take care of reading the cache line and then store the data in the message.

By now, you may have noticed that we deliberately omitted barriers in our description. Those, indeed, deserve their own space

Barriers

First of all, barriers are crucial in UMP communication to avoid that instruction reordering can result in different messages being read, for example, if the flag is set before the actual message has been written in the shared buffer. However, their usage is not the same both in the reading and in the writing. The reading required a barrier in between the flag checking and the reading of the buffer otherwise reorder is permitted thus the buffer can be read before the flag is actually set. Moreover, another barrier is needed before the flag is reset to 0 otherwise the message can be overwritten before being read. On the other side, we argue that writing does only require one memory barrier right before setting the flag. Indeed, we think that the barrier in between the reading of the flag and the is not necessary since the micro architecture should not be allowed to publish the change to the cache line before we exit the while loop which checks the flag.

7.9 Limitations

Our RPC system currently has the following known limitations:

- Not every message can be sent over every channel
  - Capabilities can only be sent across cores when forwarding over the "init" process
  - Large messages can only be sent over direct channels. They cannot be used with forwarding channels. (NB: This is handled directly by the RPC system, as a direct channel will be created when a large message is being sent)
- Channels are not cleaned up when a domain exits. This means that the channel will still be checked for new messages. However, this is not a problem, as there is never a new message arriving.

- Cyclic dependencies are not supported (i.e. two server handlers that are calling back into each other)

Chapter 8

Shell

Author: Matteo Oldani

8.1 Introduction

In this chapter of the report, I'll go through the design and implementation of a basic but working shell for our operating system.

As you know (better than us), the shell is a key component of an operating system because represents the first step for the integration and usage of multiple parts such as the file system and the network which potential can be unleashed only by allowing interaction with them. Indeed, it really gives life to the OS.

Moving to the technicalities, I'll start explaining how I ported and set up the UART driver into the userspace. This process took some time since I had to make some changes to the well-known services which every core should have. After discussing the difficulties and possible improvements that can be done to the protocol that manages the UART device, I'll start presenting the shell itself with its peculiarities. In the end, I will prepare the ground for my colleagues' projects and present possible improvements for my work.

8.2 UART in user space

Since we already had implemented a serial server running in a separate domain, I decided to use that domain as the home for the UART device driver. In general, all the individual projects that have required new servers (file system and networking), run in a separate domain.

As explained in the book, the first step is to get the correct capability referring to the correct part of memory which holds the registers for the UART device. Discussing with my teammates, we realized that we all had to somehow get different parts from the same capability. We decided to add a new well-known capability to all the processes which will need the "dev_cap", namely, serial server, file system and networking. This has been handled in the function that creates the c-space of the new process.

8.2.1 Preparing the UART

The rest of the process to set up the UART is handled in the serial server domain, in particular, all the functions can be found in serial_device.c.

Following the steps described in the book, I had to retype the correct part of the dev capability accordingly to the UART specification and also get the capability in charge of handling the interrupts. With the GCI went also the first design choice. I decided to decouple the handling of interrupts from the serial interface from the request that any domain can make to get or put chars to the serial device itself. To obtain this behaviour, all the interrupts are handled on a new waitset which is only used by the UART itself. More on the protocol will follow but the general idea is to store all the char that derives from an interrupt in different "sessions", then every session can be read whenever the client asks for a char.

All the information needed to handle the UART is allocated inside a struct, called serial_device, which is stored inside the serial_server. Indeed, is the latter in charge of initiating the device and setting up the handler for the interrupts. After that, any char typed will be registered.

8.2.2 The client perspective

On the client-side, all the domains have access to the aos_rpc calls to put and get a char. This requires me to talk about the serial server(s) reorganization. During previous milestones, we developed our os to work with up to 4 cores and always tried to separate all the domains as much as possible which we believed to be the correct and natural way to develop an os based on the assumptions made in barrelfish. For this reason, we always had 4 different copies of well-known servers, all of them accessible from every core.

This approach does not hold anymore for the serial server. Since the UART interface is unique, to avoid unexpected behaviours while setting up a serial server capable of directly interacting with the UART on every core, I would have to think about a synchronization method/protocol. This felt wrong from the beginning for at least two reasons. Firstly the whole idea conflicts with the concept of serialization where there should be a single point ordering everything. The second one takes into account performances. While evaluating our messaging system we noticed that direct UMP channel communication does not take more than direct LMP channel communication. Thus it is faster and easier to establish a direct channel between the domain that wants to talk to the serial server than have to synchronize, which implies message sending, between a different serial server which has to decide who can use the serial device.

The final decision was to only spawn a serial server on core 0 which will serve all the system.

8.2.3 The protocol

I will now further explain the protocol that handles all the accesses, with special carefulness in writing, to the UART.

Writing

Starting with the printing system, I had to ensure that only a putchar request at a time can arrive at the UART itself. This can be easily achieved using a dedicated lock for the writing which is stored inside the struct that holds the state of the UART. I decided to not implement any session aware mechanism to avoid concurrent issues with the putchar RPC call since anyone can either use printf calls or send_string RPC calls which ensures that the string is serialized. This serialization is guaranteed by the usage of locks at the server-side (in particular the write_lock, which is only released after the whole string has been printed).

Reading

To move onto the reading system, I need to explicitly clarify which problem I'm trying to address. The entities that should be able to safely ask for input from the serial port are:

- Different domains on the same core
- Different domains on different cores
- Different threads in the same domain

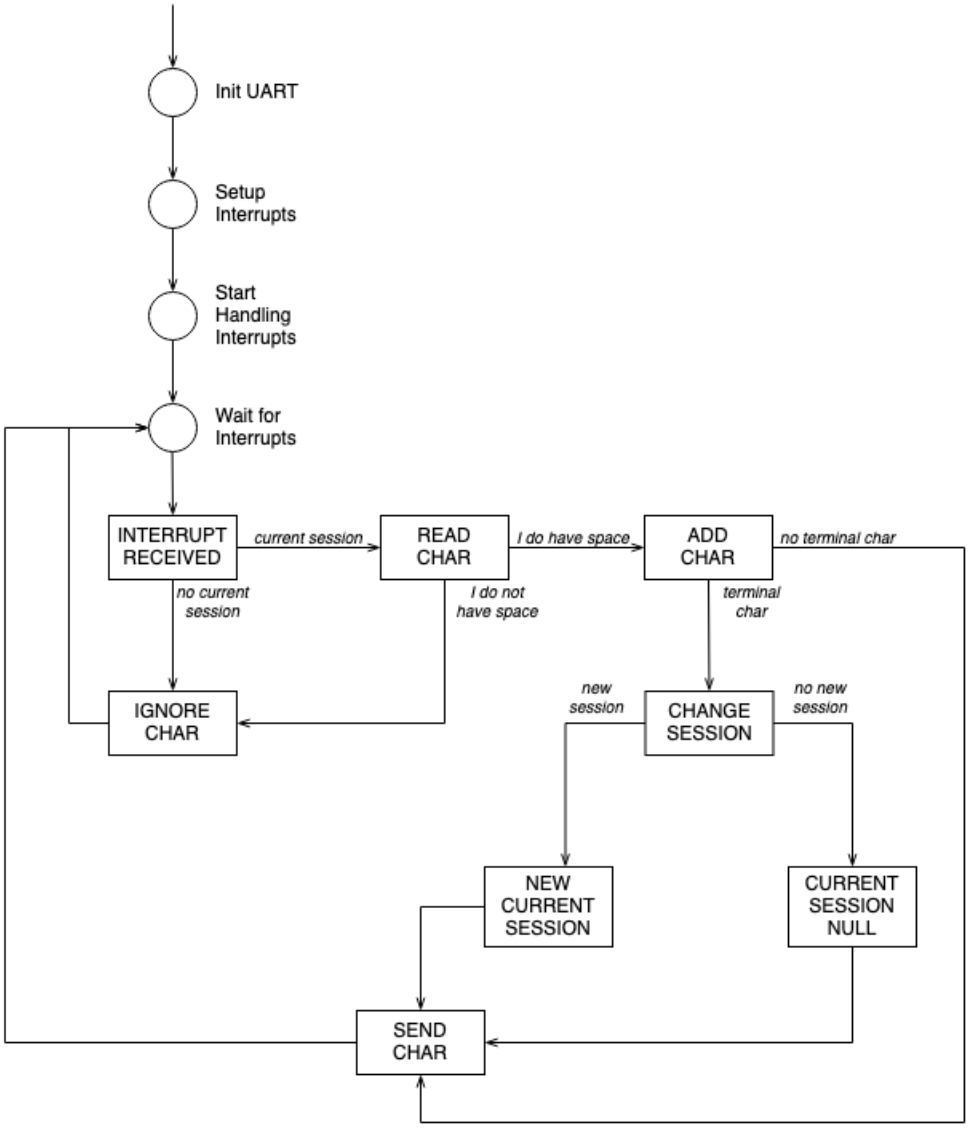
However, as requested in the book, we should have a granularity up to the single character from the client's perspective. This will require some sort of session management.

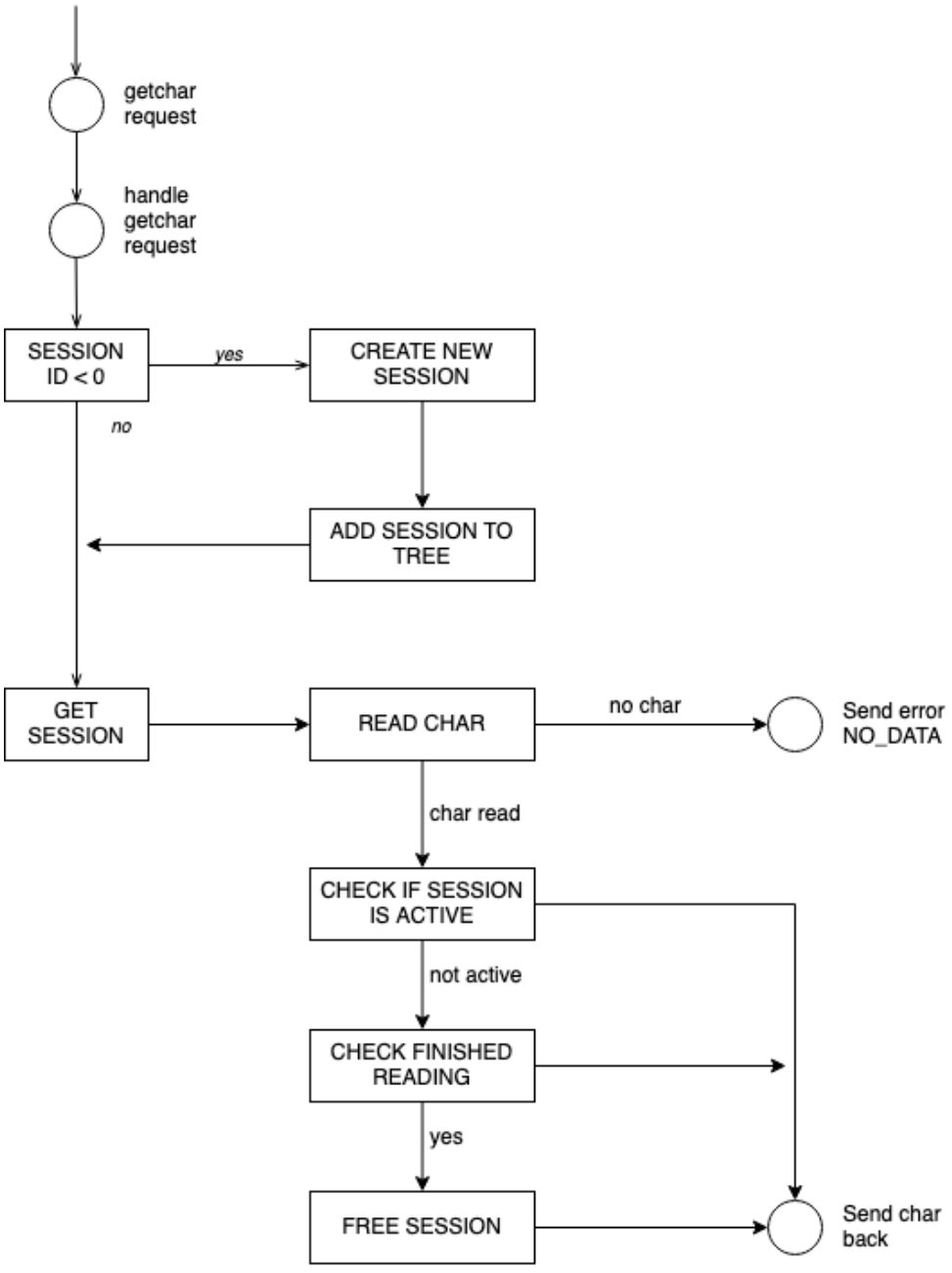
The difficulties in implementing straightforward session management based on the mutual exchange of session ids arise when I realized that we cannot delegate the bookkeeping of the sessions itself to the clients. They, indeed, are not allowed to specify anything more than the channel and the character where they want the result to be stored. Moreover, clients should not have to directly deal with a low level (fragile) session management protocol. In the following lines the solution I came up with.

Sessions

First of all, what do I consider a session? In my protocol, a session is a struct which contains: ID which is ensured to be unique. As I will mention later, this unique ID is guessable and this might represent a security issue as I will discuss later; a buffer, which I decided to make "BASE_PAGE_SIZE" long since in theory that the maximum amount of data that a program can require; indexes that hold the position for the last written and last read char in the buffer; a flag which specified if the session is still active; the lock which needs to be acquired before reading from the session; the "rb tree node" which will make the sessions a tree;

UART SIDE


Init UART
Setup Interrupts
Start Handling Interrupts
Wait for Interrupts
INTERRUPT RECEIVED
current session
READ CHAR
I do have space
ADD CHAR
no terminal char
no current session
I do not have space
terminal char
IGNORE CHAR
new session
CHANGE SESSION
no new session
NEW CURRENT SESSION
CURRENT SESSION NULL
SEND CHAR


SERIAL SERVER SIDE
getchar request
handle getchar request
SESSION ID < 0
yes
CREATE NEW SESSION
no
ADD SESSION TO TREE
GET SESSION
READ CHAR
no char
Send error NO_DATA
char read
CHECK IF SESSION IS ACTIVE
not active
CHECK FINISHED READING
yes
FREE SESSION
Send char back

8.2.4 UART Perspective

From the perspective of the UART, we can either have a "current session" or nothing. In the first case, all the interrupts will cause a read from the device which will be subsequently stored in the current session. In the second case, they will be ignored but still read. The reading even in case of no session available is required to keep the UART alive since I noticed that if I was not consuming the char written then no other interrupts will be generated. Unfortunately, I did not have enough time to dig deeper into it and I was quite satisfied with the solution so I could move on.

A special mention has to be made in case we have a current session and we read one of the terminator char. In that case, the session will be marked as "not active" and removed from the status of "current version". After that, if another session is waiting to be served, this session is set as "current session" otherwise the "current session" is set to NULL.

8.2.5 Client perspective on Protocol

Now I will explain how the client can and have to interact with the aos_rpc library to properly use sessions. First of all, I've decided to not support multiple sessions across different threads in the same domain and more on this choice will be said later. Thus, the session state for the domain is saved inside the aos_rpc file. The session-id, which by default is set to -1, is included in the messages used in the context of aos_rpc_serial_getchar. This will allow every domain to specify from which session they want to read. The session id is then updated with every response from the serial server which will sequentially assign a new session id. The serial server is also in charge of resetting the session id to -1 as soon as a terminator is read from the session.

To address potential concurrency issues between different threads that want to read from the UART, the session id is protected by a lock which can only be acquired when the session id is set to -1 and will be released only when the session is over (session-id reset to -1). Operations on the session id are allowed only to the thread which owns the lock.

8.2.6 Comments on the design choices

Now that the protocol is defined, I'll comment on it, especially on why I settled on this solution. At first, I intended to support multi multiple sessions in the same domain to allow multiple threads to read from the UART (concurrently). To implement that solution, I had to specify another rpc_call to allow the session management at the thread level. This approach looked flawed to me in two aspects. Firstly, the UART will handle session line by line thus it would not be possible in any case to write char in different threads at the same time. Thus the bottleneck of the idea will then be the serial device itself and then adding complexity to the system was not justified. Secondly, outsourcing the session management opens a more error-prone system. Moreover, talking about the security of the system, giving the ability to specify the session id could be a huge vulnerability since every process will, in theory, be allowed to read every session. To conclude on the security aspects, I know that the "stealing session" problem is still in place since there's no authentication whatsoever that binds

a specific session to a particular domain. Thus if a malicious domain crafts a message with a different session id then it can read the buffer. Unfortunately, due to time constraints, I did not have time to investigate further on this topic.

8.2.7 Drawbacks and compromises

I would argue that the major compromise (that I'm aware of) is in the conjunction between the serial server and the RPC call. To not change the sending infrastructure I decided to make the serial server non-blocking for the library call. This is done by allowing the server to respond with an error (NO_DATA) if nothing has been read from the UART. On the other hand, from the user perspective, the rpc_call needs to be blocking, at least in my design idea. Then, the rpc_call is forced to be blocked by re-asking to read a char until it gets one. A better solution can be implemented by re-thinking how the serial server responds to messages. I could have deployed a tree where all the getchar requests that cannot be served due to lack of data can be stored while waiting for a char to appear. Then, while handling interrupts to add chars to the sessions, I could have searched through the tree and served a corresponding request. This would also require a restructuring of the server itself. Indeed, the handler is now triggered by a closure and after the return, it will automatically trigger a reply. To integrate the proposed improvement is necessary to decouple the handling of the request with the sending of the reply. In theory, I would need a server able to return without sending a reply, at least for the getchar case.

Another drawback of my approach is related to session/thread handling. In the current implementation, if the thread which is bound to the current terminates (for example due to an error) without ending the session, then that session will remain active forever basically blocking all the other possible new sessions. This issue can be mitigated by setting a fairness policy which can switch to another session in a round-robin approach if the time spent waiting for a new input is greater than a predetermined threshold. However, I will point out the root of the issue as a lack of communication when a process ends either because it is killed due to a failure or because it terminates itself after completing its process. Since this problem is strongly related to another compromise made during the design of the shell, it will be further discussed later.

8.3 The Shell

I will now present the shell itself and the changes that I had to make to the system to accommodate it.

As with all the other components, the shell runs in its separate domain and for its functioning, it does not require to be a server. Thus we can see it as a normal domain which makes usage of the RPC calls to interact with the system.

Since the main goal of the shell is to be able to interpret commands and execute them, the input system is a good place to start.

8.3.1 The input system

The shell can be seen as an infinite loop over the "aos_rpc_getchar" function. I decided to not handle any buffer manipulation in the serial server, indeed, all

the char read is stored in the buffer and read by the shell which will maintain its separate buffer representing the current command line.

The shell parses every char and behaves differently based on the input. At the moment not all the "special" characters are supported but the user is allowed to use backspace to correct the input and tabs. A special case is reached whenever an ending char, such as "enter", is read. In that case, the input of the user is parsed and executed. How this is possible will be the topic of the next paragraph. To conclude the reading system, after a command has been executed, the buffer will be cleaned and the shell will be ready to receive new inputs. As already said while explaining the UART interface, the serial interface is not responsible for the echoing of the char read. Indeed, it won't even be able to work properly since the special chars are, in theory, handled by the shell. Thus, the shell itself, every time a char is received has the duty of echoing it to the screen so that the user can have feedback on what is being typed. This is achieved using the aos_rpc_putchar call.

8.3.2 Command execution

After a terminating char is read, the command line buffer needs to be parsed and eventually, a command is executed. If the buffer is empty, then it is easy since nothing is done. On the other size, a full buffer needs to be divided into multiple arguments where the first one has to be the name of the command that should be executed. Once the cmd line has been parsed and tokenized (into argv and argc) three possible paths are available. The command inserted by the user can be one of the built-in commands, a binary on the system or nothing. The last case will cause the printing of an error while the other two cases will be better explained below.

8.3.3 Well-known Commands

The shell natively supports a list of built-in commands that can be executed. Those are basic commands which can help test the system and demonstrate the possibility of the shell itself. I decided to not create these commands as separate binaries but directly inside the shell. This decision aims to avoid the overhead of spawning a new domain even for simple tasks which are meant to be done by the shell itself. The next paragraphs will describe those commands.

Basic commands

Inside the file basic.c you can find all the very basic commands such as: clear: this command is used to clear the screen hello: this command is used to display the hello message echo: this command is used to echo the input of the user to the console help: this command is used to print the list of built-in commands with a brief description

ps

This command is used to print the list of all the processes running on the system at a given time. Thanks to the already implemented RPC calls this command has been easy to develop. Indeed, for every core, I need to retrieve the PIDs

of all processes (with aos_rpc_processes_get_all_pids) and then asl for the name of those processes (with aos_rpc_process_get_name).

Even if this command was easy to implement on the shell side, it was really helpful and definitely not painless on the server-side. Indeed, I discovered some bugs in the list we were using to track all the processes. In particular, we forgot to make the list thread-safe at first and then we had to make some adjustments to the list creation mechanism which is now different depending on the core. The second issue derives from the fact that the spawned server needs to fake the first part of the list since all the well-known servers are spawned by init. The problem was that not all the well-known servers will now be spawned on every core.

A compromise which is still in place is the non-atomicity of the ps command. This can cause some errors in case a process terminates in between the get_pids call and the get_name. This process is transparent to the user who will always see the correct list displayed but we add to handle carefully that error.

kill

This command allows a user to kill a specific process on a specified core. To work it needs both the core and the PID of the process to be killed, both of which can be easily retrieved from the "ps" command.

Talking about the implementation, this command relies on an RPC call that causes the spawn server to act as if the process has made an EXIT_REQUEST. The spawn server will indeed revoke the capability of the dispatcher which will kill the process.

More details on the capability operations will be provided in the related chapter.

run_memtests

This command is provided for demo purposes. As described by the command itself it will take in input a number of bytes, allocate a buffer of that size and then touch all the bytes to first write some random numbers and then read them out. All the process is printed on the screen as proof of concept. If page-fault handling or allocation is not working, either an assertion will fail or possibly even a deadlock depending on where the issue was. If nothing goes wrong the shell will prompt you for the next command.

test_threads

As the one before, even this command is provided for demo purposes. In particular, it aims to show that user-level threads are working.

The command will require a number as an input parameter and it will then try to start that amount of threads. Every thread will simply use printf to acknowledge that it is alive (15 times).

Even if this looks like a simple operation, there's more going on when threads use printf concurrently. Indeed, the concurrence of RPC calls (such as get_channel and send_str) is tested because the threads need to talk with the serial sever. Moreover, the session mechanism described in one of the previous chapters is heavily used and hopefully proved to be working.

If the command ends without errors, or worse, deadlocks (which should be the expected and normal outcome) then the threads should be working great and the shell will prompt you for the next command.

Oncore

This command has a simple task. It will spawn the specified binary on the specified core with the specified arguments. Everything specified, should it be easy, shouldn't it? Partially.

This is the command that requires more changes in the spawning and communication system. But let's start describing the desired output.

As mentioned before, a user should be able to ask the shell to spawn a binary on a specific core. This can be easily done by calling the correct aos_rpc which we are using since milestone 3, namely aos_rpc_process_spawn (and aos_rpc_process_spawn_target). And it actually worked out of the box.

But, we did not develop a blocking RPC call thus the shell will not wait until the end of the spawned processes but only until the RPC call returns indicating that the process has been spawned (or not in case of an error). Thus we decided to implement a blocking RPC call which will be described right after the commands.

To conclude with oncore, it is now possible to spawn a process and make the shell wait for its execution to be terminated or insert an "&" at the end to specify that the shell spawns it in a detached manner and only waits for the non-blocking RPC call to finish.

File System commands

Since our os will have a file system, we decided to implement some commands to easily perform actions on that from the shell. Thus we decided to support the following commands:

- cat
- cp
- ls
- mkdir
- rm

As the names suggested, those commands are simpler implementations of the more powerful Unix commands. Talking about how they were implemented, we were able to use standard functions such as fopen and fread since the implementation of the file system is POSIX compliant and glued to the libc.

8.3.4 Network commands

Those types of commands were not intended to be present in our prototype decision. However, after my teammate developed the network and we merged we realized that was easy to implement at least one command which can be used as a small proof of concept that the network is there and (hopefully) alive. Indeed, we decided to support the "ip" command which simply shows the IP

address of the machine. This command is implemented with a simple RPC call which directly contacts the network domain to get back the information.

8.3.5 How to add new commands

In light of the design choice of making basic shell commands as built-in functions of the shell itself, I tried to make the code as expandable as possible. The final goal was to allow further development or expansion of the shell to be painless.

Thus I decided to create a common function definition that all the built-in commands need to follow. I voluntarily left the signature of those functions as general as possible, indeed they all get the full tokenized cmdline. This allowed me to create both a list where all the pointers to well-known functions are stored and one where all the names of the function are stored (in alphabetic order). The list of names can be used to check if a command inserted by the user is present among the well-known ones. In that case, the function to be called can be easily found in the list.

In practice, both lists are implemented as arrays since all the well-known commands are predetermined. In regards to how efficient this solution can be, the list should remain small in size but in any case, the lookup can be done using a dichotomic search since the array is ordered.

8.3.6 Blocking spawning

As mentioned above, we implement a blocking RPC call to allow a blocking spawning of a process. This feature is particularly relevant for the shell but can be considered a need for every process in the OS which might want to spawn and then wait.

I will divide the process into two major parts: How to not block the server while blocking the client How to acknowledge the end of a process.

The first one required both a change in the send and receive functions implemented in our message manager and in the spawninfo struct. Starting from the latter, the spawninfo now holds a flag which indicates if the spawned domain is observed or not by another domain. Alongside the flag, is also stored the struct that holds the information to contact the observer once the domain exits (the chan_msg_args struct). That information is set by the spawned server once the request by the clients arrives. Moreover, in case of a blocking spawn request, the server will set the RPC of the channel to NULL. This is required to make the change in the rpc_message_manager effective. Indeed, in case a NULL RPC channel is found the send is never executed. This will result in the server (the spawn one in this case) not being blocked while the client is blocked since it is waiting to receive a message which won't come.

Moving on to the second part of the process, we might need to find a way to answer the client who is still waiting for a response. This requires the process to be able to contact the spawn server and notify them that it is exciting. It is finally arrived at the time to implement the libc_exit function present in the init.c file. I decided to make all the processes contact the spawn server once they reached that function. They will send an EXIT_REQUEST message which will not give any response back to them. However, on the server-side, once the exit message arrives the server will check if the process was observed by someone or not. In case it was not it will simply remove the process from the list of

the running processes and return since the client does not expect any response. But, in case the process was actually observed, the spawn server will use that "chance to respond" to send the message to the observer which channel is saved in the spawinfo of the process itself.

To summarize the process: process A can ask the spawn server to spawn a process B and register itself as an observer. This will cause the server to not immediately reply to process A. When process B terminates, the spawn server will be contacted and it will finally reply to process A which should still be listening.

To conclude on this topic I will mention a drawback of this approach which is related to why it was difficult to create a blocking rpc_getchar call in our setting. The process explained works because we are allowing only one observer at a time. In particular, what was difficult before was to send a message from the server without having received a response. This compromise is still in place but it's avoided by using the request from process "B" to reply to process "A".

8.3.7 Outside the Well-Known commands

We are back on the main flow of this chapter, in particular, we have now to treat the case our command is not among the well-known. Then we need to check if it matches a binary either in the multiboot or in the filesystem. Both cases can be checked by querying the spawn server with a "check_binary" request.

If the binary is found, then it will be spawned on the current core which in our case is always core 0. Otherwise, a "command not found" error message will be printed by the shell which will be ready to receive e new command.

8.3.8 Performance Analysis

As for the spawning mechanism (see Section 5.5), we also evaluated the performance of the shell. To be more specific, we evaluated the built-in "oncore" command with blocking and detached spawning for core 0 and 1. In all cases, we spawned a process with an empty "main" function ten times and measured the time it takes for the shell command to finish.

Figure 8.1 shows the time it takes to spawn the process in blocking mode - meaning that the command only returns when the spawned process finished. As the spawned process immediately returns, this is the closest we get to having the raw overhead of the spawning mechanism. Figure 8.2 shows the time it takes to spawn a process in detached mode - meaning that the shell immediately returns after the RPC call to the spawn server and does not wait for the spawned process to finish.

In both variants, spawning on core 1 is faster. This is as expected, because the shell runs on core 0, so there are two cores working in parallel when spawning on core 1. It is also visible that the time needed for spawning is linearly increasing in the number of spawned processes. This is also an expected result, as channels of exited domains do not get cleaned up (see Section 7.9), so the channels are still polled, leading to the linear runtime.

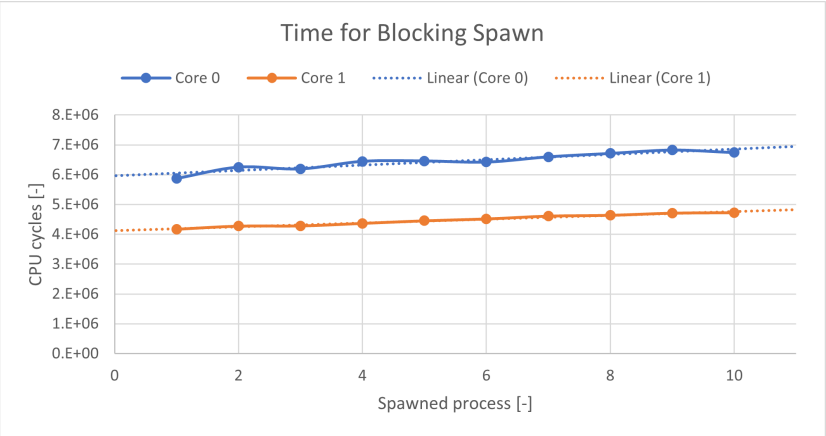


Figure 8.1: Execution time for spawning an immediately exiting process from the shell in blocked mode. The shell waits until the spawned process is finished.

8.4 Conclusion

This concludes our journey in moving the UART driver into user space and using it to build a working shell. From now on the ground is set and all other projects can make use of it both for development and testing.

As a final remark, I would add some personal thoughts on the work done. First of all, I tried my best to not interfere much with the whole system that we have developed since on one side it would have caused tons of problems during the final integration and on the other side, it won't have been coherent with the choices made as a group. Even if I'm convinced with the approach this choice introduces some compromises as discussed in the sections above. To conclude, the part that I struggled with the most was fast typing. At first, I thought that there were concurrency issues between the serial interface and the multiple sessions. For sure at the beginning there were some possible deadlocks, however, after fixing those and having restructured all the locking systems to be more conservative, the problem looked only mitigated. Indeed, with really fast typing or with a busy core, the shell was still getting stuck. In the end, I discovered that the issues were related to interrupt handling. In particular, every time the function responsible for handling new incoming char was not done before another char was typed the interrupts stops coming. As later explained on moodle, it is expected behaviour. Unfortunately, I did not have time to change the protocol to completely defeat the bug but I managed to fix the "busy core" issue which tracked down to a "non-yielding" busy loop. At the current state of the OS, the shell supports a fairly fast typing (which means that I'm not able to reproduce the bug only with simple typing).

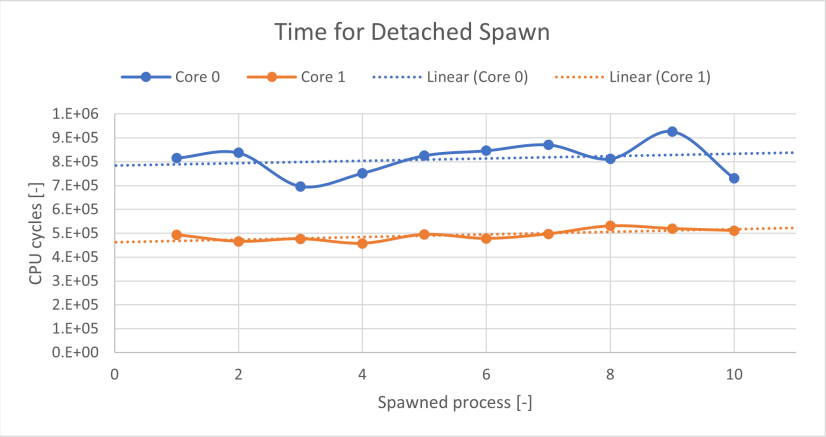


Figure 8.2: Execution time for spawning an immediately exiting process from the shell in detached mode. The shell returns directly after the RPC call to the spawn server and does not wait until the spawned process is finished.

Chapter 9

Capabilities Revisited

Author: Jasmin Schult

9.1 Correctness

The "correct" management of capabilities is vital to ensure that the layers of abstraction and isolation, both between kernel and user space and between different user space domains work correctly and the execution of applications is safe and secure.

To be able to build a "correct" capability management system, we first need to define what exactly "correctness" means in this context.

I think there are two different aspects to this notion of correctness (although I wouldn't be surprise if there were a more beautiful and universal way to look at them that combines them into one - I am admittedly not very good at finding such universal abstractions...):

First of all, there is the notion of a correct **state** of all the capabilities in the system. Secondly, we have to specify when the distributed execution of a capability operation is considered correct, i.e. we have to define what exactly we want a "revoke" or "retype" to do in this distributed setting.

I am making the distinction between this "operational" and the "system state" correctness because I feel like the two are on slightly different levels of abstraction - The isolation and safety properties the OS is supposed to provide between kernel and userspace and different user domains mainly hinges on the capability system never entering an "incorrect state". In contrast to this, operational correctness **should** ideally imply system state correctness, which requires that our operations as well as the enforced consistency model are sensible.

9.1.1 Correct System State

It makes sense to express the correctness of the capability management system's state in terms of invariants, which I will do in the following enumeration. There is not much to say about their choice (except some justification why I didn't make any changes), they just correspond to the invariants proposed by the book. However, the operations with respect to which they are supposed to be "invariant" are worth discussing:

1. The **natural capability system invariants with respect to syscalls**: Different cores must not have conflicting views on the *purpose* of a certain region of physical memory - the most disastrous case likely being a region mapped into a user domain's address space as a frame on one core while being used as a Cnode on the other. Thus, quite naturally, the **non-overlapping except between parent and children** and **compatible capability types invariants must be upheld** as discussed in section 13.9. One may be tempted to think that a certain leniency as with the ownership invariants (see below) can be tolerated; for instance, allowing a retype on one core if all the conflicting capabilities on the other are in the process of being deleted. This is, however, a fallacy: the "usage" of a capability may not necessarily be prevented just because it happens to be in the kernel's "in_delete queue" - consider for instance a page table mapping: as long as it is not "cleaned up", the mapping remains active, allowing the corresponding domain access to the memory region in question. Therefore, these invariants must be upheld with respect to every single syscall. An example consequence of this is that it is not sufficient to synchronize only the "mark phase" of revocations (at which point both cores will agree that the capabilities are in deletion), but also the sweep phase must be completed on both cores before the revocation target can be safely retyped!

2. the **ownership invariants with respect to protocol transactions**: I adopted the three capability ownership invariants proposed in the book. I think the ownership concept is very natural because we also divide the initial physical memory regions between the cores. Additionally, some way of breaking the symmetry in the protocol is crucial - while there may be approaches that could work without, their liveness guarantees would be diminished (if both cores attempt simultaneous conflicting retypes at the same time, both MUST fail unless symmetry is broken) and quite surely the implementation complexity would skyrocket.

   These invariants cannot be upheld at the syscall granularity - certainly not if we allow ownership transfer because there is no way to ensure that ownership is updated atomically for all cores. Also, in the case of revocations, it may be that the owner's delete queue is quicker in deleting its copy than the queue on the other core (as it happens, remotely owned copies are usually deleted during the mark phase so this generally does not happen).

   Therefore, the ownership invariants are only "invariant" with respect to "protocol transactions" - they can be temporarily violated while a revoke or ownership transfer inducing operation is in progress.

9.1.2 Operational Consistency

Intuitively, what we strive for is that the operations in the distributed capability management system behave "like" their single-node system counterparts. Thus, the most natural consistency model to enforce on distributed executions is **sequential consistency**.

Of course, it may very well be that a weaker consistency model could also make sense. However, I believe that especially for interleavings of "revoke"

with "capability transfers", anything less than sequential consistency doesn't make sense: otherwise, it could happen that a transferred capability survives a revocation of its subtree and as a result revoke would become "best effort" only, meaning that after the revocation completes (and no further operations are performed on that core), there is no guarantee that no remote descendants of the revocation target exist anymore and thus the flag "remote descendants", once it is set, could never be cleared again.

Therefore, the operational correctness my protocol strives for is sequential consistency.

9.1.3 Supported operations, restrictions and requirements

Apart from upholding the invariants described above, there are a few more knobs that can be turned - both with respect to the supported operations as well as additional, more implementation-oriented requirements.

In the following, I will list all of the different such "knobs" that I have been considering while working on this individual project. Additional explanations why (and in which combinations) these are relevant for the resulting protocol complexity will follow in subsequent sections. Additionally, I marked the choices I ended up with in bold.

1. **The number of supported cores**

   There are two different options here:

   (a) **Two cores**

   (b) Many cores

   This is because supporting only one core is not interesting (we couldn't call it "distributed" capability management in that case), and exactly two cores is the only other special case that exists. This is because with exactly two cores, we get a total (there is only one receiver, namely the single "other node" in the system) and causal (because the channels between the two nodes are FIFO and therefore it cannot happen that messages sent by the same node are re-ordered) message ordering for free. Given the limited time, I was forced to restrict the supported cores to two (because the message ordering properties considerably simplify the protocol!) - more justification and example scenarios that I deemed too complex to handle can be found in section 9.1.4.

2. **Ownership transfer** This is a simple yes or no choice:

   (a) **Yes**, ownership transfer is supported

   (b) No, it is not supported.

   But what exactly do I mean with ownership transfer?

   For the purpose of this discussion (and in future sections), it helps to view the capability system as a forest, whereby each tree is rooted at a particular initial RAM capability defined in the bootinfo struct (this is not entirely accurate, after all there is also a parent type for RAM capabilities, namely "PhysAddr", which represents the unique node of the capability system).

The question of whether or not we allow ownership transfer then boils down to whether or not the ownership within such a tree is static (after all the initial root capabilities have been assigned to the cores).

I think that support for ownership transfer is important to support all possible process interactions one might want to perform in an operating system - imagine for instance some domain on core A requests the spawning of a "child" process on core B and shares a frame with it (maybe to transfer some initial data etc). It would not be very user friendly if this frame were deleted from the child's CSpace as soon as the parent exits.

Additionally, the capability management protocol becomes quite trivial in that case:

(a) retyping will never require any coordination on the owning core (because it is aware of all potential conflicting capabilities). If a remote desires a retype, it will simply ask the owner, which will then send back a corresponding foreign capability (or not if the retype conflicts).

(b) there will never happen any revocations across "ownership boundaries" because every capability in a capability tree of the forest is owned by the same core.

(c) any issues caused by in-flight capabilities (see the discussions below) can easily be resolved by synchronizing over the owner.

Thus, my protocol is supporting ownership transfer, which can happen as result of the following operations:

(a) Via a send operation that also transfers ownership of the sent capability.

(b) As result of a delete last copy on the owning core, the ownership of the capability can be transferred to the remote core if it still has a copy. (This is the case that would allow the child process described above to continue to use the frame even once its parent exits).

(c) As result of a retype operation invoked on a remotely-owned capability, the remote core can transfer ownership of the retyped capability (if the retype succeeds) to the remote core.

Note that no operation exists that would allow a core to request that ownership of a remotely owned capability be transferred to it. This is to prevent the remote core from requesting such ownership transfers and subsequently perform a revoke on the now owned capability. In my opinion, the owning core should get to decide when and if it it deletes a capbility, unless the owner of an ancestor decides to revoke said (which makes sense because the owner of the ancestor has precedence).

3. **Which capabilities can be sent** Here again, there are two choices:

   (a) only the owner can send a capability to another core

   (b) **any core can send a capability to another core**

Whether or not this has any impact on the complexity of the resulting protocol is dependent on some other choices (such as ownership transfer and number of supported cores).

At first glance, one may think that in the two core case, there is no need for a core to send a capability to its owner, because said will already be in possession of said capability according to our ownership invariants. However, just because a particular core has got a copy of a capability, this does by no means imply that all domains have access to it. Therefore, sending a capability to the owning core does make sense, because there is no guarantee that the receiving domain will already have a copy of it!

4. **synchronization message handling** This point is a bit of an odd one out because it is on a different level of abstraction: all other knobs talk about supported operations in the capability system, while this one is on the implementation level. Nevertheless, I found that it has very much influenced my protocol design so I still want to mention it here.

   It is about how much context a node needs to consider when handling a synchronization message (i.e. a message that is used to coordinate a monitor operation) and whether or not a response to the message needs to be stalled until another event occurs.

   (a) **Simple handling**, i.e. no stalling and no context needs to be considered when handling a synchronization message (with the sole exception of the REVOKE_SWEEP, where the REVOKE_SWEEP_DONE answer is registered as a callback of the delete step framework.)

   (b) Complex handling: Synchronization messages may need to be stalled and/or context of ongoing transactions need to be considered during their handling.

   It is quite obvious that the first choice eases implementation considerably and greatly simplifies the reasoning about liveness. However, as discussed below, the support of certain combinations of operations can require that the second option be used.

For your convenience, here again the summary of the different options I have identified, without the walls of text in-between...!

1. **The number of supported cores**

   (a) **Two cores**

   (b) Many cores

2. **Ownership transfer**

   (a) **Yes**, ownership transfer is supported

   (b) No, it is not supported.

3. **Which capabilities can be sent**

   (a) only the owner can send a capability to another core

   (b) **any core can send a capability to another core**

4. **synchronization message handling**

   (a) **Simple handling**, i.e. no stalling and no context needs to be considered when handling a synchronization message (with the sole exception of the **RevokeSweep**, where the **RevokeSweep** answer is registered as a callback of the delete step framework.)

   (b) Complex handling: Synchronization messages may need to be stalled and/or context of ongoing transactions need to be considered during their handling.

9.1.4 Why supporting many cores is hard

You might be asking yourselves: why is she even thinking about supporting arbitrarily many cores?

The reason is we have been working with four running cores since milestone 5 and we have invested a lot of effort into the generalization of the rpc framework to support arbitrarily many cores. During that process, I became very accustomed to thinking about "this local core vs all the other cores out there". As a result, it did not occur to me in the beginning that I could to dial back to cores to two, I immediately started trying to solve the capability management problem for arbitrary many cores.

Of course, in hindsight when reading over the project description again, it is very clear the support for arbitrarily many cores is NOT required - there are quite a few passages that talk about **the** other core and **core 0 and core 1** specifically - unfortunately, these "hints" were lost on me so maybe a more explicit warning might help the future, similarly unwary students to spend their energy a bit more wisely...

But let's get back on track: why is the problem so much harder with arbitrary many cores? The reason, as already mentioned above, is that we cannot rely on a total order message delivery.

As a result, I believe that no protocol exists that supports arbitrarily many cores and ownership transfer without needing to resort to "complex handling of synchronization messages", i.e. in terms of the options defined above, 1b and 2a implies 4b.

This claim results from my not being able to find a "simple synchronization message" handling solution for the following two scenarios: (Of course, it can very well be that I am wrong and I just wasn't able to find the solution - if so, please let me know, I am very interested to hear what you think!)

Both have the following initial state (depicted in figure 9.1): there is a capability C owned by core 0, as well as a descendant D owned by core 1 (which happened as a result of some kind of ownership transfer operation, which we assume is supported by the system).

Then, one of the following things happens:

Scenario 1 Core 0 now initiates a revoke on capability C, while core 1 simultaneously sends capability D to core 2.

If synchronization messages are neither stalled nor additional transaction context is considered, it may happen that core 2 receives and handles the revocation synchronization messages for C before capability D arrives,

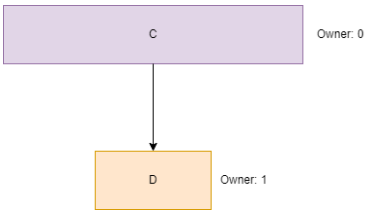


Figure 9.1: The initial setting for the problematic many core cases

and will thus accept the capability D once it arrives without any means of realizing that it should have been deleted during the revocation.

See figure 9.2 for an illustration of the issue.

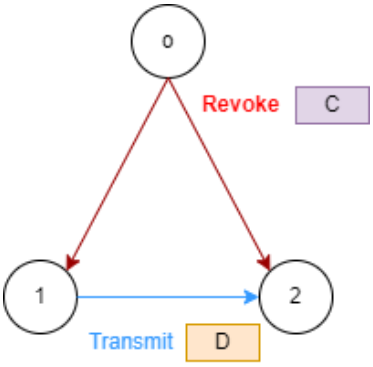


Figure 9.2: A depiction of the unfortunate revoke - transit interleaving: the arrows indicate operations and point from initiator to the responder(s). The number next to the operation indicates the logical sequence in which the handling of the requests happen at the responders.

Scenario 2 Core 0 now attempts to perform a retype of capability C that conflicts with D, while core 1 performs an ownership-transfer send of D to core 2 and subsequently deletes its copy of the now remotely-owned capability D.

Again, without complex handling of synchronization messages, it could be that core 2 receives core 0's retype request before it receives capability D and thus answers "yes". The retype request for core 1, however, is delayed until after it has deleted its copy of D and will thus also result in a positive answer. Thus, core 0 will perform the retype even though the conflicting capability D still exists in the system.

See figure 9.3 for an illustration of the issue.

Of course, there may exist an easy solution despite "complex" synchronization message handling, but the only approaches I could think of seemed quite complex both in terms of implementation and specification (as soon as messages are being handled out of order and/or past operations are considered, the TLA+ model becomes too complex to specify as well as too large to run the model check on it):

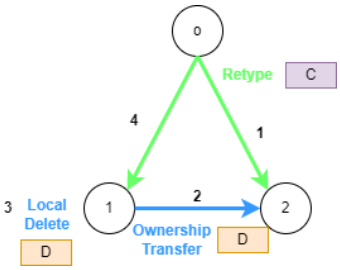


Figure 9.3: A depiction of the unfortunate retype - ownership transfer interleaving: the arrows indicate operations and point from initiator to the responder(s). The number next to the operation indicates the logical sequence in which the handling of the requests happen at the responders.

- **Vector clock like approaches** Potential solutions might be to assign increasing per-core sequence numbers to each operation the corresponding core orchestates. Each core then also tracks past operations as well as a vector clock that indicates for up to which sequence number all operations of the corresponding core were handled.

  We can then either:

  - Include the clock when sending a capability and have the receiver re-apply the operations that it has already performed but the sender of the capability had not.

    This only solves scenario 1 because we can simply "re-apply" the delete, but of course with a retype answer this doesn't work because core 0 already received a positive answer from us so it won't know to expect yet another, negative one.
  - Include the clock when sending an acknowledgement for the send back to the receiver - meanwhile, the sender stalls the handling of all synchronization messages except for the "transfer acknowledgement". It then checks the clock against its own and waits for all the operations it is missing to arrive, such that it can provide the correct negative retype answer / inform core 2 that the sent capability should be deleted

- **Ensure revoke/retype never happen while a capability is being sent** An alternative solution could be to ensure that revokes and retypes are never initiated during the send. This could be accomplished by adding an additional "prepare" synchronization step before either the send operations or the revokes/retypes. Cores will only acknowledge this prepare once the potentially conflicting operations are completed and they will not initiate new such conflicting operations until the in-progress one completes.

I hope that with these examples, you would agree that a protocol with support for arbitrarily many cores would have been a feat (while doubtless very interesting and educational) not easily accomplished with the given time constraints.

9.2 Design process or TLA+ to the rescue!

After finally restricting myself to two cores and with the goal of designing a protocol with "simple" synchronization messages, I sat brooding over different difficult scenarios for a while to get a feeling for the nature of such a protocol (if it indeed existed).

However, it is clear that brooding can only get one so far and that the human mind, while quite adapt at thinking up a few problematic cases is hopelessly limited when it comes to a principled exploration of all the possible problematic interleavings.

So I was very glad for your hint of creating a specification of the protocol in TLA+: it was immensely helpful, not only for designing a robust protocol but also for my own sanity: my brain has the annoying tendency to start "model checking" my designs of distributed protocols when I am trying to sleep (and occasionally also in my dreams, causing me to suddenly startle awake because I found a bug in a twisted (and nonsensical once awake) dream version of my protocol).

With the help of TLA+, I could delegate this "model checking" to TLC and thus gain sufficient confidence in my protocol that the sleepless nights subsided again.

I will now describe my final protocol design, while highlighting the design mistakes I would have made if not for the TLA+ specification.

The section after that will be dedicated to the TLA+ model itself, with special focus on its limitations and simplifications compared to the real system I ended up implementing.

9.3 The protocol

(I have been pondering for quite a while how to best present the protocol I ended up with and unfortunately haven't been able to come up with a truly satisfactory approach, but I still hope this section is (at least partly) understandable.)

In this section, I will first describe each operation individually. Each of these descriptions is comprised of a numbered sequence of steps of the format:

Node label: (**Trigger**)

whereby the node label indicates whether the initiator (INITIATOR) or the responder (RESPONDER) is executing the described step while the trigger describes an event(either the receipt of a message or the completion of the delete queue) that triggers the step.

Note that the trigger specification is redundant for almost [1] all operations : each step is executed in the given order, and the next step is always triggered by the receipt of the message sent in the previous step by the other node (or an initial client request). I am nevertheless making these triggers explicit to allow for easier cross-referencing with the TLA+ specification and my implementation.

[1] except for the last revoke step case which happens only once both the message was received and the delete queue has finished, and the last retype step which only happens under certain conditions

For the reasoning about why certain steps are required and must be performed in that way, I also need to clarify the meaning of a "step" in terms of distributed protocol executions:

The protocol assumes that each such step can be considered atomic, i.e. that all the possible protocol executions are semantically equivalent to an interleaving of such steps.

Furthermore, all client requests (which trigger the first step of an operation on the initiator) are handled sequentially, which means that at any given time, a node is initiator to at most one operation.

9.3.1 Send capability

This is the simplest operation of the protocol; however, one important aspect to keep in mind for the revocation discussion is that we don't allow capabilities to be sent that are marked for deletion.

1. INITIATOR : (cross-core RPC msg with capability C)

   check that C is not locked or marked for deletion
   set remote copies of C
   determine owner of C
   send (description of C)

2. RESPONDER : (description of C)

   create C based on received description

9.3.2 Send capability with ownership transfer

Once we start transferring ownership, things get a bit more complicated:

1. INITIATOR : (cross-core RPC msg with capability C)

   check that C is owned, not locked and not in deletion
   lock C, determine if it has local descendants
   send (description of C, has local descs)

2. RESPONDER : (description of C, has local descs)

   create C based on received description while ensuring:
   owner of C is set to self
   remote descs flag is set according to received "has local descs"
   no inconsistent states are visible to holders of existing copies
   send **OwnershipTransferAck**

3. INITIATOR : (**OwnershipTransferAck**)

   change owner of C to remote
   set remote copies flag
   unlock C

One may ask why it is not sufficient to just directly change the owner in step 1 and thus avoid the additional **OwnershipTransferAck**. The reason is that this would allow the following unfortunate sequence of events to happen:

1. We change the owner of C and send the ownership transfer message

2. We delete capability C (which is now possible without coordination because it is remotely owned)

3. Meanwhile, before handling our ownership transfer message, the remote sends us capability C back, but with the outdated ownership information that indicates that we are owner. Thus, once we reconstruct C from that message, we end up with contradictory ownership views for C, which violates our invariants.

Another aspect about this operation that is not immediately obvious is why we are required to lock capability C in step 1. This is to prevent local retypes of C while the ownership transfer is in progress: If C had neither local nor remote descendants in step 1, it could still be retyped locally, i.e. without monitor involvement, thus causing the "has local descendants" flag sent to remote to become outdated. The remote may thus mistakenly perform conflicting retypes on the now-owned capability C.

9.3.3 Revoke

1. INITIATOR : (Client Revoke Request on C)

   check C owned and not locked
   lock C
   send **RevokeMark(C)**

2. RESPONDER : (**RevokeMark(C)**)

   pause delete steps
   mark entire subtree of C for deletion
   send **RevokeMarkDone**

3. INITIATOR : (**RevokeMarkDone**)

   pause delete steps, register callback when delete queue done
   mark entire subtree of C except C for deletion
   resume delete steps
   send **RevokeSweep**

4. RESPONDER : (**RevokeSweep**)

   register **RevokeSweepDone** to be sent once delete queue completes
   resume delete steps

5. INITIATOR : (**RevokeSweepDone**)

   clear remote copies and descendant flags and unlock C

6. INITIATOR : (**RevokeSweepDone**, delete queue complete)

   reply to client.

One of the most interesting aspects of the revoke operation is the locking on the revocation target. Let's first take a look at why it is required in the first place:

The issue with revocation is that while the sweeping is in progress, the capability subtree rooted at the revocation target may be temporarily inconsistent and thus retypes that cause a temporary conflict could happen.

For example, the following may happen:

- the revocation target C has a child capability D that is shared with the remote, but there are no remote-only descendants and thus C's remote descendant flag is not set.
- during the mark phase, the remote copy flag on D is cleared to prevent the delete step framework from unnecessarily negotiating the deletion of D with the remote (and we therefore "forget" that D is shared)
- the sweep phase on the initiator deletes capability D more quickly than the sweep phase on the remote
- before D is deleted on the remote node, the domain holding the revocation target could thus perform a local retype on C that conflicts with D, thus violating our invariants.

The locking on the revocation target prevents this conflicting retype on D (because local retypes will fail if the source is locked and will be handed to the monitor for coordination).

The question now is: why do we already lock in the first step if these inconsistencies only emerge during the sweeping? The reason for this is implementational ease - locking the revocation target in the very first step has the additional benefit that it cannot be deleted before we can call the targeted revocation marking on it (otherwise we would need to handle the resulting failure).

The unlocking of the revocation target in step 5 is also worth discussing: why can't we wait with the unlocking until the operation completes? (This was actually one of the design bugs that TLC's liveness checking uncovered for me and that I am quite sure I would have missed otherwise!)

It turns out that in case of overlapping revocations, we would deadlock if we only unlocked at the very end - the delete step framework on the node that revokes the descendant could never complete because its revocation target is both locked and marked for deletion.

As a result, we have to unlock precisely when we receive the **RevokeSweepDone** message - at this point, conflicting retypes are not a problem anymore because any potential "unknown" remote descendants have been deleted. Additionally, liveness is guaranteed because in case of overlapping revocations, the node that revokes the ancestor will be able to complete its sweep operation because none of its "in-deletion" nodes are locked, which will allow the unlocking of the descendant revocation target on the other node and thus, ultimately, the completion of both operations.

The second interesting aspect is how we ensure that no in-flight capabilities survive a revocation performed of their ancestors, i.e. how we guarantee sequential consistency between revocations and transmissions. The answer to this follows from the following two observations:

1. We do not allow a capability to be sent to the other core if it is marked for deletion. Thus, the responder of the revocation can only initiate a transfer of a "to be revoked" capability before step 2, i.e. before it receives the **MarkRevoke** message.
2. All client requests (which also includes the transmission of capabilities) are serialized on each node. This means that transmissions performed by the initiator of the revocation are always sequentially consistent.

Thus, the worst that could happen is that the responder sends a to-be-revoked capability before step 2 of the revocation. In this case, the capability cannot escape deletion because it will be received strictly before the **MarkRevokeDone** message that triggers step 3 at the initiator, which correctly marks the received capability for deletion.

There is one little subtlety here that I got wrong at first and that the TLA+ model helped me uncover: originally, I had not intended to serialize the sending of capabilities along with all other client requests. Instead, I had the revoke initiator perform a "mark and mark and sweep", i.e. I already marked the capabilities for deletion in step 1 to prevent the initiating core from transmitting a "to-be-deleted" capability to the responder after step 1. The above argumentation then still applies: indeed, any capability sent by the responder before step 2 will be correctly marked for deletion **at the initiator**. The scenario that I failed to consider is that nothing is preventing the initiator from sending the capability back to the responder before step 2. In that case, the capability will successfully escape deletion by "surfing on the mark wave".

Therefore, the serialization of capability transmission alongside all other client requests is vital in that it confines capabilities on the initiating core until the operation is handled.

9.3.4 Delete Last

This operation is a bit of an odd one out - as you can see, it is the only one that is not initiated by a client request, but by the delete step framework.

Of course, there IS a delete client request - however, the initialization and finalization of its handling is just not very interesting: upon receiving it, the monitor will simply mark the desired capability for deletion and let the delete step framework do its job, to then report completion back to the client once the delete queue is finished.

For this reason, I decided not to bother you with a detailed description of these additional surrounding steps.

1. INITIATOR : (delete step encountered SYS_ERR_DELETE_LAST on C)

   pause delete steps
   lock C
   determine if C has local descendants
   send **DeletionNotice(C, has descs)**

2. RESPONDER : (**DeletionNotice(C, has descs)**)

   If (has copy of C, C moveable):
   set owner of C to self (transfer = True)
   Else:
   delete all foreign copies of C (transfer = False)
   send **DeletionAck(transfer)**

3. INITIATOR : (**DeletionAck(transfer)**)

   If (transfer):
   set owner of C to remote
   Else:
   unset remote copies of C

unlock C
mark C for deletion / delete C
resume delete steps

As described in the execution steps, there are two different ways in which a *delete last* can proceed - it can either result in an ownership transfer if the capability type allows it and the remote still has a copy of C or in a complete deletion of C from the system.

There is not much to say about the former case - it is essentially equivalent to an ownership transfer send with a subsequent local deletion at the initiator. The only interesting aspect is why we lock the capability - after all, it is the last copy on the node and held by the delete step framework, which will not perform any conflicting retypes on it. Bear in mind that this is not set in stone - it could be that the remote sends us a message containing this capability in the mean time, so locking is still required.

The latter case is also not very interesting - it is essentially a simplified revocation coordination because we know that the remote can only have foreign copies of the capability which can be immediately and easily deleted, without needing to involve the delete step framework.

9.3.5 Retype

1. INITIATOR : (Client retype request of D on C)

   check that C is not indelete
   check that retype of D is allowed locally
   send **CanRetype(D, C)**

2. RESPONDER : (**CanRetype(D, C)**)

   If (C is owned):
   success = retype D from C, transfer = success
   lock D (if retype successful)
   Else:
   success = check if retype of D allowed, transfer = false
   send **CanRetypeReply(transfer, success)**

3. INITIATOR : (**CanRetypeReply(transfer, success)**)

   If (success):
   success = retype D from C
   If (transfer):
   send **RetypeTransferAck**
   set remote copies of D (if retype successful)
   reply to client

4. RESPONDER : (**RetypeTransferAck**)

   set owner of D to remote, set remote copies
   unlock and delete D

This is the most complex operation of the entire protocol, interleavings with simultaneous operations performed by the other node can create very subtle and hard to identify issues. Therefore, let's discuss the potential conflicts and how they are avoided one by one:

Retype - retype conflicts

Retype-retype conflicts are prevented by breaking the symmetry of the retype operation in step 2: if the node is the owner, it performs the retype the remote it requesting itself and keeps the resulting retyped capabilities until the remote acknowledges the transfer of their ownership.

This ensures that if both nodes initiate conflicting retype requests simultaneously, it does not happen that both succeed because the owner of the source is accounting for any retype request it answered "yes" to, so its own conflicting retype would fail in step 3 at the latest.

Note that it has to be the owner that tracks retype requests in this manner, the remote would not be able to do it: first of all, it may not even possess a copy of the source capability to perform the retype on and secondly, owning node does not necessarily coordinate all its retypes with the remote - this is only necessary of the "remote descendants" flag in the source is set, which only happens once there may be child capabilities that only the remote is aware of.

This symmetry breaking is substantially simpler than serializing all retype requests for a specific capability over the owner, which is what I considered first:

If the owner of the capability must serialize all retypes on it, they have to be queued at the owner's monitor. This means that either we allow a node to delegate retype requests for remotely owned capabilities to their owner without waiting for them to be handled, which is problematic if the owner performs an ownership transfer at some point.

Alternatively, we would need to resort to complex synchronization message handling: while waiting for our queued retype request to be handled by the remote, we must continue to handle the retypes the remote has queued at our monitor, otherwise we may deadlock (both nodes wait for the remote to handle a retype that is enqueued somewhere in the request queue and will thus only be handled once the remote answers the current retype request, i.e. never).

Retype-ownership transfer conflicts

There are two types of retype - ownership transfer conflicts that can occur:

Firstly, it could be the result of a retype that conflicts with a capability whose ownership is currently being transferred. As we have already seen in the multicore case, the challenge is to ensure that we correctly account for the capability, even if its ownership is in limbo. We solve this issue by ensuring that the former owner holds onto the capability until the ownership transfer is confirmed. Like this, a conflicting retype is guaranteed to fail because at least one of the nodes is aware of the conflicting capability and will respond negatively to retype requests.

Secondly, and more interestingly, it could be that the ownership of the retype's source capability is being transferred.

As already discussed above in the context of sends with ownership transfer, we ensure that no local retypes (that would have required a different "has local desc" flag to be sent to the remote) can happen on the initiator by locking on the capability while ownership transfer is in progress.

However, what about simultanous retypes on the responder? There is one very tricky scenario (credits to TLC, I would never have come up with this execution!) that is the reason why we set the **remote copy** flag in step 3 if

a transfer occurred, even though the other node will subsequently delete the transferred capabilities:

Initial situation: core 0 owns capability C, core 1 has a copy of C.

- core 0 initiates an ownership transfer send, while core 1 simultaneously requests to retype capability D from capability C.
- core 0, still the owner of C by its bookkeeping because the ownership transfer hasn't been confirmed yet answers "yes" to the retype and creates its own copy of D
- meanwhile, core 1 changes C's owner to itself and acknowledges the ownership transfer to core 0
- finally, the retype "yes" arrives at core 1, causing it to retype D. If D's remote copies flag were not set, core 1 could now perform the following operations: delete D, locally retype a conflicting capability D' from the now owned C (which works because C's remote descendants flag is not set, because C didn't have any local descendants when core 0 initiated the ownership transfer). As a result, both D and conflicting D' may temporarily exist in the system until core 0 deletes its copy D, violating our invariants.

Note that of course, setting the **remote copies** flag to prevent simple deletes is but one solution - alternatively, we could always set the **remote descendants** flag at the new owner during ownership transfers, which ensures that all retypes on it are coordinated.

The cleanest solution would probably be to add an additional round of locking and synchronization to the ownership transfers, but I have decided against that because I feel like the protocol is sufficiently complex already, and every additional exchange of messages provides more room for bugs or liveness issues.

I hope that I could convince you that retyping is complex business - it turns out that on the implementation level, there is an additional challenge related to in deletion sources at the owner, that I will discuss in the section on the required changes to the kernel.

9.4 Implementation

The monitor implementation is performed by exactly two threads running in each init process:

- The monitor thread.
- The delete step handler thread.

The operation of the delete step handler thread is not terribly interesting to discuss - its implementation is mostly unchanged from the provided one, with the exception for some extra functinality to handle the freeing of memory (which is discussed in the RAM reclamation section below).

The monitor thread is implementing the execution steps detailed in the protocol description below, pretty much exactly as described. The only exception

is that the finalization of client requests (i.e. just providing the reply to them) is split away as a separate step.

This eases the implementation, because in combination with the protocol requirement that the handling of synchronization message be "simple" (i.e. no stalling, no additional context of ongoing / past operations considered), the monitor is simply working two queues:

- A queue with synchronization messages. These messages may be of two types:
  - synchronization messages sent by the remote monitor thread, such as **RevokeSweep, RevokeSweepDone, RevokeMark, RevokeMarkDone, CanRetype, CanRetypeReply, DeletionNotice, DeletionAck, OwnershipTransferAck** and so on. They are simply matched against the execution step that should be triggered by their receipt, which is then executed.
  - messages that represent steps that must be taken on behalf of the delete step framework, such as coordination for *delete last* or ram reclamation. They cannot be performed delete step framework itself: the protocol assumes that steps are atomic, and therefore we cannot allow two threads to perform steps concurrenty. Of course, a lock would have done the trick too, but since we need a queue with synchronization messages anyway, we may as well enqueue delete step tasks on it too.

9.5 The TLA+ model

You can find the pretty-printed version of my TLA+ specification in section A. I would definitely be able to write it in a cleaner, more understandable way now, so please forgive a TLA+ beginner's initial clumsiness.

9.5.1 limitations and restrictions

As you will notice at first glance, the model heavily abstracts the nature of the capability system:

- It does not attempt to model CSpaces, i.e. there is no concept of using a memory region to store capabilities to other memory regions in the model. I would argue that this should not impact the faithfulness of the model: whether such cascading deletions are performed in context of handling a single client request or multiple consecutive ones (as TLC will perform during its exhaustive model checking) should not impact the soundness of the distributed protocol, because the "separation" of consecutive operations into client requests is a purely node-local concept.
- Since the idea of capabilities that are accessible to different domain is very tightly coupled to CSpaces, domains cannot be represented either. As a result, the model is conceptually restricted to a system with only two init processes.

- Without domains, there is also not much sense in describing the notion that capabilities can be copied, because that is only relevant for the sharing of capabilities between domains. This has the additional benefit that capabilities can be modelled as sets instead of bags, and eliminates the unnecessary state space explosion that results from the fact that arbitrarily many copies could be created and deleted again without it having any relevant effect on our distributed capability management.

- Capabilities are untyped. The only really interesting type would have been CNodes and Dispatchers, because their deletion results in a cascade of capability deletions. Since the model is representing neither, it also does not make sense to represent capability types. (Note that if the protocol is broken and supports retypes with conflicting types, it will also allow overlapping child capability retypes, which TLC can pick up even if capabilities are untyped.)

Of course, not modelling domains does have some impact on the model faithfulness, because all of the system calls are always relative to the current domain's CSpace and a lot of complexity in the monitor code originates from correctly accessing and retyping capabilities located in other CSpaces. For examples of this complexity, please refer to section 9.8.

There are some additional simplifications and constraints the model needs to impose to prevent a state space explosion due to unbounded queues:

- The number of unconsumed capability transfer messages (without ownership transfer) is restricted. This is necessary because as detailed in section 9.3.1, sending a capability does not require a response from remote and therefore the operation immediately completes at the initiator. Thus, without restriction, the queue of capability transfer messages could grow unboundedly.

- As opposed to the real implementation, the model also does not make the queuing of client requests explicit. Instead, there is a step that allows the model to "spontaneously" initate the handling of a new client request whenever the previous one has completed. The reason for this is to again prevent an unbounded request queue, because clients can issue new requests at any given point in time.

9.5.2 Aspects that are faithfully represented

After reading the above section, you may ask "are there any aspects of the protocol left that she did not abstract away?". Indeed, there are:

- The different execution contexts of the monitor implementation, namely monitor thread (represented by the requestHandlers variable) and the delete step thread(represented by the deleteHandlers variable) are faithfully represented.

- The model represents the communication channels between the two inits as two (one for each direction) as sequences, where to-be-sent messages are enqueued at the tail and to-be-received messages are popped from the head. This is an accurate representation because the two monitor

threads are exclusively using the ump "init forwarding ump channels" to coordinate (regardless of whether the sent message could be considered a request or a reply).

9.6 Model checking

As I have mentioned above, the TLC model checker very much saved my skin, multiple times! The properties that I let the model checker check are provided in the beginning of the TLA+ model pretty print, in comment boxes.

Next to the capability system invariants that I have detailed in the initial section, I am also checking the following liveness property: The request handler will always eventually finish handling the client request (which in terms of the model specification means: will always eventually be in the "IDLE" state). For it that to hold, I had to mark all actions that drive the request handling forward as weakly fair, which include:

(To refer back to the initial section on correctness: note that I am checking the **state invariants** and not that the protocol yields sequentially consistent capability operations, i.e. not operational correctness because I cannot think of a way to check the latter automatically)

- handling a synchronization message
- performing a delete step
- signalling that the delete queue is finished
- finalizing a client request (i.e. sending the reply)

I think weak fairness is very reasonable for these types of actions - after all, provided that we do not deadlock (which is exactly what we want to verify) and the monitor handler or delete step thread do not crash, there is no reason why we would not eventually make progress in the request handling.

Limitations

So here comes the big "but": unfortunately, despite the precautions taken above to prevent state explosions, the only capability system I was able to exhaustively verify is one with a memory region of unit size 2, i.e. if all retypes that can be performed are performed, we end up with exactly three different capabilities in the system: the root of size two and two children of size one.

The breath-first exhaustive search in a system with unit size 3 (!) ran out of memory after 9 hours and at depth 22.

Clearly, a successfully exhaustive search in a system with only three different capabilities is insufficient to say with confidence that no protocol bugs remain.

However, I would hope that with all executions of length at most 22 in the "larger" system, we have likely explored most conflicts between concurrent operations with all of the relevant ancestor-descendant patterns that can occur.

9.7 RAM reclamation

Being able to reclaim and reuse memory is vital for anything that wishes to call itself an "operating system": if we cannot do so, we are doomed to run out of memory and fail after spawning about 40 processes (we counted!).

9.7.1 Why the existing mechanism did not work

Technically, reclaiming RAM is not explicitly part of this individual project, presumably because there exists a mechanism that (maybe?) for some specific protocols works out of the box.

However, for my particular protocol, the existing mechanism does not work correctly, indeed it may even jeopardize its correctness: During `cleanup_last`, the kernel checks if the capability that is being cleaned up has **neither local ancestors nor local descendants**. In this case, it will copy the capability into the slot provided as argument to the `delete_step` or `clear_step` syscall responsible for the call to `cleanup_last` and return `SYS_ERR_RAM_CAP_CREATED` to indicate that the capability can be freed.

This is not working correctly for two reasons:

- Without further synchronization, this *cannot* be safe for any protocol that supports **owernship transfer**: the absence of local ancestors or descendants does not at all imply that no remotely-owned capabilities exist that cover the corresponding RAM region.

  This is no biggie if only remote descendants exist: the memory server can simply perform a revoke on the capability before considering it free and poof the problem is solved.

  However, the existence of remote ancestors is a completely different story, for two reasons:

  - The other node could perform a revoke on the remotely-owned ancestor at any arbitrary point in time. As a result, the RAM suddenly disappears from the mem server's supply or, even worse, is deleted from an unsuspecting client's Cspace. Thus, the entire system could become extremely volatile, with process data or, worse, instructions, suddenly and unrecoverably vanishing.
  - What is even worse, it may break our invariants (badly)! It could be that the reclaimed capability as well as its remote ancestor were frames (because retyping a frame into a smaller frame is allowed!). If the reclamation mechanism now hands us back RAM capability, this is disastrous!

- Secondly our mm design only implicitly (i.e. without holding specific capabilities to them) stores free regions and dynamically retypes from the "root capability", which it initially received via mm add, when a client requests ram. Thus, the root capability will always exist (since the mm uses it to retype) - actually I would claim that this has to be the case unless either the memory server is privileged and can use "monitor create" to merge neighbouring free regions back together or we just fragment the memory further and further, without ever merging free regions back

together (which, needless to say, is a slower death than without support of memory reclamation, but ultimately just as fatal).

So, as a result, the only case in which the memory reclamation mechansim will spring into action is when we delete the last capability to a memory region that was initially awarded to the other core, i.e. if a remote ancestor (namely the aforementioned root) is guaranteed to exist!

9.7.2 How to fix it

Clearly, the mechanism needs adaptation to our *always retype for the root* mm design as well as additional synchronization to avoid the "remote ancestor" problem.

First of all, I made the kernel aware of the mm's root retyping: I added an additional flag to the mdbnode type called **root**, that is always set to false except for the capabilities that init transfers to the mem server.

I then adapted the reclamation mechanism to only return a cleaned-up memory region as a canditate for freeing iff:

- its parent capability has the **root** flag set
- it has no copies (neither remote nor local) and no local descendants.

If these conditions are met, `cleanup_last` will return the newly created RAM capability along with the return code `SYS_ERR_ASK_REMOTE_ABOUT_FREE`. This will cause the delete step framework to tell the monitor thread (by enqueuing on the described synchronization message queue) that it should ask the remote node if it still possesses any capability that covers the memory region in question. If the remote replies in the negative, the capability is handed to the mem server for freeing.

Of course, you may ask yourself: why is she not simply checking the remote ancestor and remote descendant flag to check if freeing is allowed? First of all, the capability deletion in the kernel is not propagating the remote ancestor flag correctly. Secondly, both remote ancestor and remote descendants, if set, can be false positives so a synchronization mechanism would be required anyway to handle these cases.

Note that because the ram capability is still created, we are technically still violating our capability system's invariants (remote ancestor is a frame issue). However, as long as the capability is not retyped (which it cannot be because the only copy is safe in the delete step framework's care), this does not really cause any issue because RAM capabilities can only be retyped and (at last as far as I can tell) do not have any other side-effects on the system.

If I had had more time at my disposal, I would have liked to change the interface of the delete and clean step syscalls to instead pass in a capability struct pointer into which the kernel could store the information about the capability instead of truly creating it, in order to avoid this "technical" violation of the invariants altogether.

Unfortunately, there is one additional somewhat dirty trick that I had to apply for this procedure to work correctly:

The owner of the temporarily created capability needs to be set to the remote core. There are two reasons for this:

- It could be that the remote core indeed owns this capability (note that it does not correspond to the one we deleted but it is instead its "RAM-type" parent) and could send it to us while the free question is being asked. If we then innocently create the cap based on the provided description, we will encounter an owner mismatch and abort. Thus, the owner needs to be set to remote because if anyone owned the capability, it would be the remote.

- The second reason is the freeing at the mem server: the mm will take the received capability, delete it and adapt its bookkeeping to consider the region to be free once more. If we don't set the owner of the capability to remote, this deletion will result in a monitor call (because cleanup last wants to return a candidate region for freeing (!)) and the entire dance begins again.

So with this mechanism, we can free memory regions in response to deleting capabilities on the core that owns the corresponding root region.

But what if a remotely owned capability outlives the capabilities on the core that owns the corresponding root region? To also address this case, I further extended cleanup_last:

If a capability is deleted that is the last covering the region of ram (i.e. exactly the same condition as the old reclamation mechanism employed), cleanup_last will also return the corresponding RAM capability along with the return code SYS_ERR_INFORM_REMOTE_ABOUT_FREE. In this case, the monitor thread will identify the capability, check that its region is still uncovered (we could have received new capabilities from the remote in the meantime) and if so inform the remote. The remote can the hand the region to the mem server for freeing if it is only covered by the root region.

9.7.3 Limitations

There are two fundamental limitations to this RAM reclamation mechanism.

First of all, the lack of verification: I am not entirely confident that it works correctly in all scenarios; adding it into the TLA+ specification for model checking would have been very helpful. Unfortunately, the time was not sufficient to extend the model appropriately, and even if it had been it would have resulted in even more state and more possible steps, rendering the checking of a meaningfully large system even more impossible.

Secondly, RAM can still be lost by the mechanism. This has also been true for the old version: if a client splits a capability A into multiple children and then deletes some of them before deleting A and some of them after, the memory covered by all the children deleted before A will be lost.

The reason for this is simple: in the worst case, the client has deleted every other child before deleting A. If we wanted to reclaim all the memory freed by A's deletion, the delete step framework would need to provide as many as $\frac{|A|}{2}$ fresh capability slots.

Thus, to avoid losing memory because of strange deletion orderings, we would need to additionally introduce some form of garbage collection at the mem server: it would need to periodically query if its used regions are indeed still used. Unfortunately, the time has not been sufficient to implement such a

feature and thus we currently reply on our client domains to sensibly delete the memory awarded to them.

9.8 Changes to the Kernel

This section summarizes the necessary changes to the kernel.

- syscalls to the monitor to acquire a copy of the capability in another domain while also locking it, and the reverse operation (unlocking and deleting the copy). This allows for *atomic* monitor operations with respect to that capability, because other domains that have access to it cannot delete or retype it in the mean time. This is useful because otherwise, there is always the danger of the monitor copying out a capability to check its status and update remote relations, especially during send operations, only for the domain who owns the copy to delete it, causing the monitor to be stranded with a capability that it may not be able to get rid of easily (for instance because the remote copies bit is set and thus deletion requires coordination).
- a modification to the revoke marking that allows the revocation target (and thus any copies on the owner core) to be locked upon entry (the regular deletion of copies fails if the capability is locked)
- the additional root flag in the mdbnode structure to allow for RAM reclamation
- additional syscalls to check if a particular RAM region can be reclaimed: one for usage on the core that does not own the initial region (checks that region uncovered completely), the other for usage on the core that owns the initial region.
- a slight modification to the retyping checks, to prevent retypes on capabilities that are marked as being *in_ deletion*
- removal of the *not in deletion* assertion when receiving a capability (because the monitor has no way to check that) - since this can only happen with complex deletions (and not when revoking), it was only the specific copy that should be deleted, not *all copies* so we can perform the delete when we receive a new copy from the other node.

Chapter 10

File System

Author: Bowen Wu

10.1 Introduction

File system is an integral part to any non-trivial operating systems. It largely extends a computer's capability of storage and are sometimes used to back up the limited memory in case of OOM. Previously, our system must only launch programs that have already been pre-baked into the multiboot modules. This severe lack of flexibility and extensibility prevents us from spawning more useful programs. Therefore, in this milestone, we extend our system with a file system.

We implemented our file system based on a FAT-32 partitioned SD card. FAT-32 file system is composed of a file allocation table (namely FAT) and data regions. The file allocation table contains a series of 32-bit entries, each of which holds the index of the next cluster. The data region consists of clusters, where each cluster belongs to at most one file/directory. For each file/directory, we only need to know the index of its first cluster and then we can query the whole file by looking up FAT. An example file read procedure looks like the following.

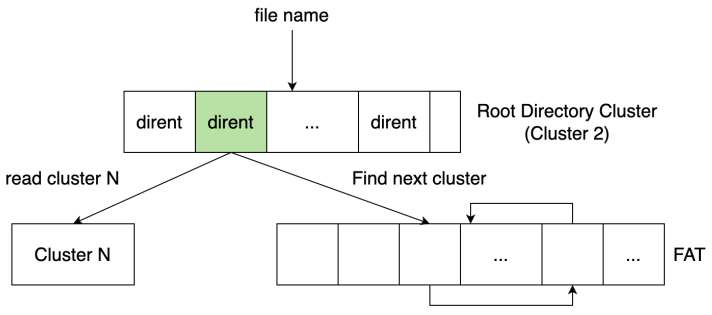


Figure 10.1: File read in FAT-32 based file system

1. Given the file name, read the root directory cluster. Check the directory entry there one by one until a match is found. From the matching entry, we can know the first cluster index of the file.

2. Read the cluster's content.

3. Look up the FAT to see if there are more clusters associated with this file. if yes, find the next cluster and go to 2.

10.2 Architecture

We established a dedicated file server to serve file system operations as is the common philosophy of a micro-kernel operating system. The file server is on the bootstrap core and processes requests from all cores. A single instance of file server has the following advantages.

- Easiness. The host controller (i.e., SDHC) requires sequential accesses. If we let multiple cores access SDHC, we need to synchronize them. Besides the SDHC, the FAT-32 data structures also need to be consistent and therefore can only be accessed sequentially. Because of such sequential access requirements, we opt for a single file system server design. We argue that this single instance design will not deteriorate the performance and on the contrary, it may improve the performance as we will discuss later.

- Better global monitoring. With just one file server, we have a single view of opened files and directories. This helps us coordinating the accesses to files from different domains. For example, if a domain wants to delete a file opened by other domains, we can readily tell the deletion should hold and wait.

- Better performance. By having only one core talk to the SDHC, we can efficiently take advantage of caching. On the contrary, if all the cores can read from and write to the SDHC, caching would be more difficult and likely to be inconsistent. Especially, when using the SD card whose latency is prohibitively high, caching is vital to the performance as we will show in 10.6. And since accessing SDHC is sequential by nature, having multiple instances of file system servers will not help improve the performance.

Having such observations, we designed the file system stack as shown in Fig.10.2.

10.2.1 Layer-1 SDHC driver

Layer-1 is the provided SDHC driver that interfaces the SD card with MMC commands. In the given implementation, it reads and writes a 512-bytes block at a time and busily wait for the command to finish. This has an obvious inefficiency – a file system usually read at a larger granularity. In FAT-32, files are arranged in clusters (which contains 8 blocks). Only having a per-block-read interface means we have 8x latency when reading a cluster. Even worse, the file allocation table is usually read into the memory while initializing the file system. But because of the slow driver, it is impossible to read the 3 MB table within reasonable amount of time. This renders the cache layer (Layer-2) extremely crucial for good performance.

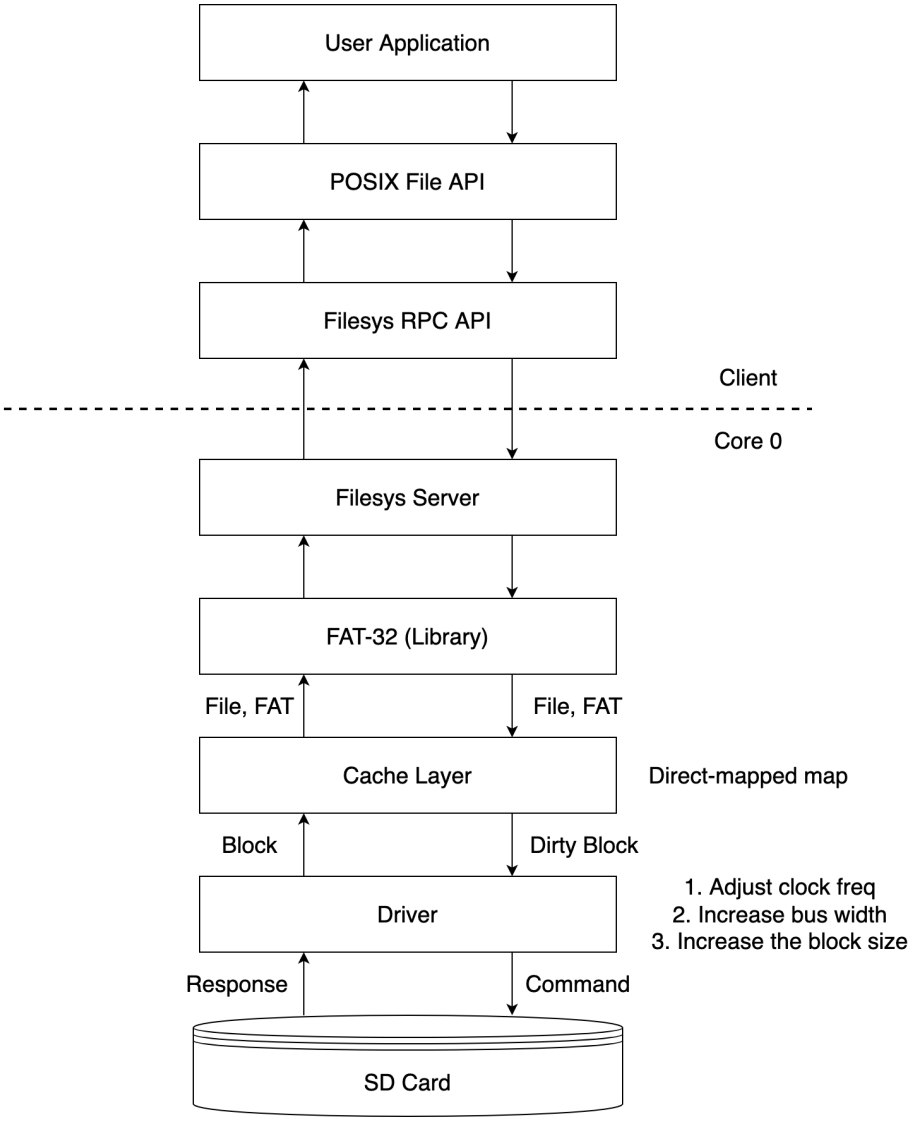


Figure 10.2: File system architecture

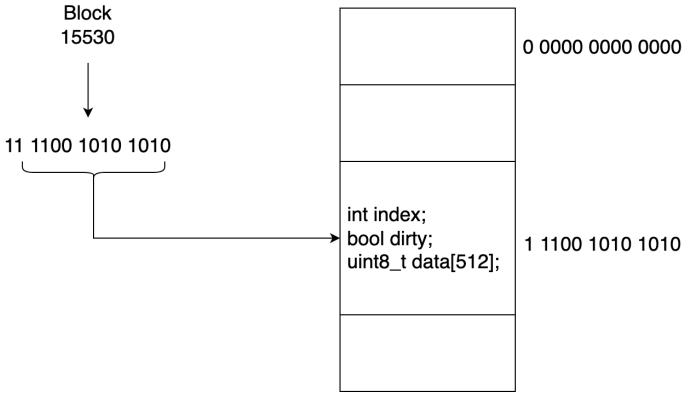


Figure 10.3: Cache example. The last 13 bit of the block index is used as the cache index.

10.2.2 Layer-2 Cache

We deployed an direct-mapped write-back cache for data blocks from and to the SDHC driver. When a read or write happens, it would first check if there is a cache hit. For each cache entry, it is associated with a dirty bit to indicate if the memory and the disk holds the same content. When there is a cache conflict, a dirty entry will be flushed whereas a clean entry will be overwritten. The cache is particularly useful for frequently accessed blocks, e.g., the file allocation table, root directory etc. As we are only experimenting with only a handful of files, conflict or capacity misses are rather rare. But for a full file system, the cache should be set-associative.

10.2.3 Layer-3 FAT-32 standard

Layer-1 and Layer-2 constitutes the storage part of the file system. Starting from Layer-3, we will build the FAT-32 abstractions and file system operations. Layer-3 enforces the storage layout and on-disk data structures to be compliant with FAT-32 standard. It implements the file system functions in a FAT-32 fashion. For example, when a user asks to create a new directory, Layer-3 would allocate a directory entry and a cluster with "dot" and "dotdot" entries filled in. In the implementation of Layer-3, we constantly need to check if a file exists or locate a directory which means FAT table and root directory data are frequently fetched. The cache layer (Layer-2) becomes very helpful in providing quick access to such hot data.

To meet required functions, we developed the following functions that bridge the FAT-32 specification and file system operations.

```
fat32_read_file   // read file content at a given offset
fat32_read_dir    // read directory entry at a given offset
fat32_write_file  // write to a file at a given offset
fat32_create_file // create a new file or a new directory
fat32_rm_file     // remove a file/directory
fat32_fileinfo    // return information of a file/directory
fat32_read_elf    // read the ELF binary
```

It is worth mentioning the optimization we have done for reading and writing files. Even though a file is stored as a chain of clusters, while reading or writing, we compute the relevant blocks based on offsets and read/write length and only read/modify those blocks. In this way, we avoid reading or writing the whole cluster each time. In case of writing with cache, we only need to mark such relevant blocks as dirty and thus avoid unnecessary flushing.

For directory deletion, we *recursively* delete everything in the directory and make sure all disk space are properly reclaimed.

For creation of new files or directories, we do not use a sophisticated way to index free clusters. Instead, we linearly scan the directory entries and FAT until we find a free slot. The reason is that building up index will require reading in the whole FAT which takes prohibitively long time. Our way makes sure that all the allocated clusters or directory entries are close to each other. In this way, we can achieve better look-up or read performance.

10.2.4 Layer-4 File system server

In Layer-4, we construct the user-space file system server which serves the RPC requests from the user to manipulate files. It keeps a global view of which file/directory are opened across the whole system. It is also used as a cache for file/directory meta information so that a subsequent open request to the same file will not need to go to the disk check if the file exists.

For such purposes, an important internal data structure of the server is the open directory-entry table (ODT). An ODT entry has a one-to-one mapping to a file/directory on the disk and keeps its status information. It keeps track of how many domains opens it (namely, reference count) so that we will not delete an "active" file/directory. Another piece of information in the entry is the position of the file/directory on the disk. With this, we can quickly jump to that position without having to search from the root directory over and over again. ODT organizes the entries in a red-black tree, which helps us quickly retrieve the correct entry given the file/directory path. Such an ODT also helps us implement commands like `lsof`.

```
// representation of an opened file/dir
struct odt_entry {
    struct rb_tree_node node;
    char*  path;
    size_t refcount;
    bool   is_dir;
    int    first_clus;
    struct fat32_dirent dirent;
    int    dirent_clus; // The cluster that contains the dirent
    int    dirent_off;  // The offset of the dirent in the cluster
};
```

Interaction with file systems usually involves transferring large amount of data over RPC channels. LMP channels is a bigger problem because by default it only transfers 32 bytes of data at a time. Fortunately, we implemented the extra challenges from the LMP chapter so that we can transfer up to 4096 bytes of data in one go. However, we could still face the risk of surpassing the limit. If

that is the case, the RPC library will abort the send with an error. Luckily, we solve this issue on the client side by enforcing a maximum bytes of read for each RPC request. For UMP, the ring buffer essentially allows arbitrarily lengthy messages so the message size is not a problem.

10.2.5 Layer 5 and beyond

Layer 1-4 reside only in the file system server domain on the bootstrap core. Layer 5 and upwards are libraries that every domain on all cores have access to. Layer-5 implements the client-side RPC functions to interact with the file system. Client RPC calls are listed as below.

```
aos_rpc_open_close
aos_rpc_read_file_target
aos_rpc_open_dir_target
aos_rpc_close_dir_target
aos_rpc_read_dir_target
aos_rpc_write_file_target
aos_rpc_create_remove
aos_rpc_fileinfo
aos_rpc_read_elf
```

Layer-6 implements the functions that will be glued to the C standard libraries, including open, read etc. Such functions are essentially wrappers of the RPC calls on Layer-5. In addition, it assigns file descriptors and keeps track of the open files for the current domain. Last but not the least, the user interacts with a POSIX-compliant file system interface, e.g., fopen, fclose etc. This interface is already provided.

10.3 Design decisions

Besides the design decisions discussed in the architecture section, we also encounter the following decisions.

How should concurrent accesses to a file be coordinated?

Since only one server domain can actually access the file system and interact with the SD card, read and write operations are automatically atomic. However, this does not mean the read and writes from different domains are serialized. When a domain makes a modification to a file, other domains will not be notified and it may have stale meta data, e.g., file size and cursor. For example, if both domain A and B open file C and if A truncates the file, then B's cursor may be no longer valid as it may go out of range. Since there is no hard requirement on bahaviors under such scenarios, we decided to implement the following mechanism. The cursor of a file, along with the file descriptor is local to the client domain of the file server.

In case of read or write, if the cursor is out of the scope of the file, we return zero bytes. When setting the cursor with `lseek`, we will first fetch the latest file size from the file server and calculate the new cursor position accordingly.

What if a domain tries to delete a still-opened file used by other domains?

Since we keep a reference counter for each file, we can easily tell if a file is opened by some domains. In this case, the deletion will fail if the counter is larger than zero. The caller should retry the deletion after some time. This contrasts to the Linux design decision where the deletion will succeeds while other processes can still write to the deleted file as long as they open it before the deletion. We believe this would require us to implement logical deletion and garbage collection which complicates the implementation and in the meantime does not bring much benefits to the users.

What if the user forgets to close the file before the program exists?

Because of the reference count in the ODT, it is important to make sure users will explicitly close the file before the program exits. Otherwise, there will be zombie reference counts which prevent a file from deletion permanently. For arbitrary user programs, it is impossible to force the user to comply. Therefore, we would automatically check and close all the open files/directories kept in the domain local file descriptor table after the main program has exited (i.e., in `libc_exit`.

10.4 Loading ELF for program execution

Previously, the program ELF binaries reside in the multiboot module and are loaded into memory upon booting. This is not a scalable solution as we cannot add new programs or modify existing programs when the operating system is running. Therefore, we utilize the file system as the storage of the program binaries.

When the spawn server receives a request to spawn a binary, it sends a request to the file system server to asks for the image of the binary. If the image is has not been loaded to memory, it will allocate a frame and loaded the ELF into the frame. After loading, the file system server will respond to the spawn server with the capability of the ELF frame. In this way, we can reuse the loaded ELF images and ignore the maximum message size constraints on LMP/UMP channels.

Loading an ELF is different from reading from a file because ELF is typically larger in size and we always want to read its entirety. Given this, two optimizations are in place.

1. Unlike reading files where we must copy contents to a user provided buffer, we directly provide the physical address of the ELF frame to the SDHC driver. This saves many memory copies.

2. We mark disk blocks that correspond to ELF images as non-cacheable. Instead, we have a dedicated cache just for ELF images. This is because ELF images are large in size and reading it may lead to substantial cache conflict/capacity misses.

10.5 Efforts to improve the device driver

As the provided device driver is too slow to be useful for meaningful tasks, e.g., loading ELF for execution. We attempted the following improvements.

1. Adjust the clock frequency by setting smaller divisor, i.e., dvs and sdclkfs. This does not bring any improvement.

2. Using a larger block size than 512 bytes. In this way, we reduce the number of transactions we need to fetch a cluster (of 8 blocks). Unfortunately, this is not possible because according to the SDHC controller specification, the maximum possible block length is preset in the CSD register. And by looking up that register, the maximum block length of our SD card is only $2^9 = 512$ bytes.

3. Use multiple block read (CMD18). We attempted to preset the number of blocks to be read by CMD23, however sending this command always gives us timeout. Alternatively, we can send a CMD12 to stop the multiple block read. We have not found a way to make this work.

4. Do not specify the block length every time before a read or a write. The given implementation explicitly set the block length by sending an extra command to the SD card before every read and write. This is not necessary because the default block length for the SD card is already set to 512 bytes. By eliminating this, we manage to obtain marginal speed up.

5. When waiting for a command's response, instead of sleeping, immediately call `thread_yield`. This prevents the thread from "sleeping in" and missing the command's response. This significantly speeds up our block driver by more than 5 times.

We also found a bug in the provided SDHC driver. When the `sdhc_send_cmd` waits for response to the command, it checks (`tc || cc`), which means as long as the transfer (of data) is complete **or** the command is complete, the `sdhc_send_cmd` will return. This makes it possible that while the command has completed, the data for reading/writing has not arrived. We encountered this bug after we improve the performance of driver because the `sdhc_send_cmd` returns much faster than before, leaving data transfer incomplete most of the time. We fix it by the following patch.

```
...
if(is_read || is_write) {
  if(tc && cc) break;
} else {
  if(tc || cc) break;
}
...
```

This makes sure when `sdhc_send_cmd` returns, the data is always ready.

Together with a cache layer introduced before, we can squeeze more performance by pre-warming the cache.

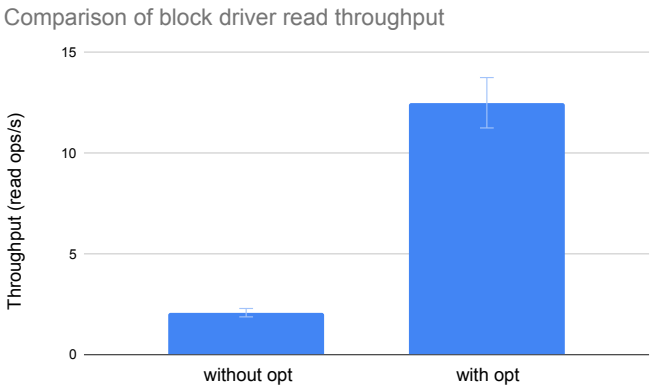


Figure 10.4: Read performance measurement of the SDHC driver.

10.6 Performance Evaluation

In this section, we evaluated the performance of our file system. We first measure the read performance of the block driver itself (before and after our optimization). Then, we report the end-to-end file read performance.

For measuring the block driver read performance, we call `sdhc_read_block` to read 100 blocks and measure the time. We compare how the read throughput (measured by read ops/s) between optimized and unoptimized versions. The result is shown in Fig.10.4.

For measuring end-to-end file read performance, we uses `fread` to read 512, 1024, 2048, 4096, 8192, 16384 bytes of data on core 0 and core 1 respectively. Comparing to just reading from the driver, reading a file will make rpc calls (via LMP or UMP). The result is shown in Fig.10.5 and 10.6.

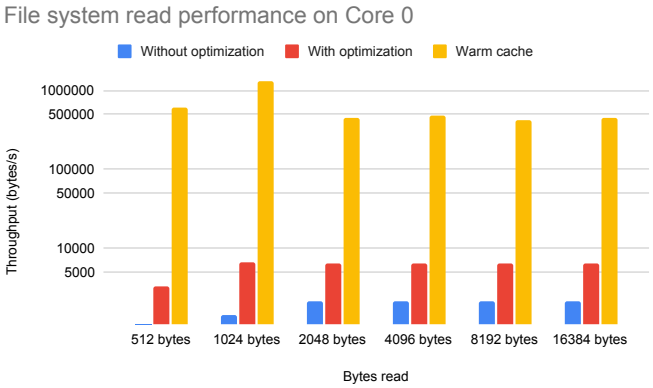


Figure 10.5: End-to-end performance measurement on core 0.

The results shows that the improvement to the block driver also substantially improves the end-to-end file read. For different read sizes, the throughput almost remains the same except for 512 bytes due to RPC overhead. A strange thing

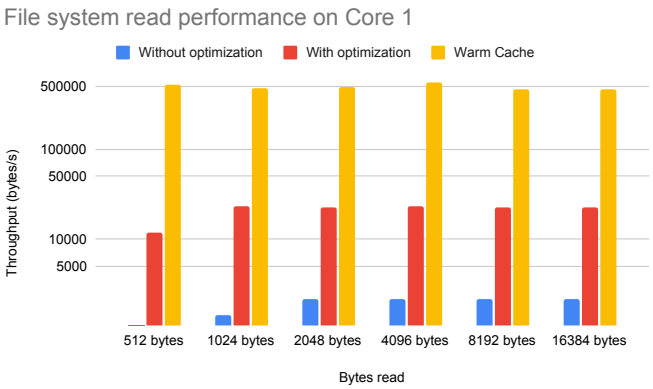


Figure 10.6: End-to-end performance measurement on core 1.

we noticed is Core 1 has a much better performance than Core 0. The root cause is that the UMP request handler thread, out of some reasons, read from the block driver much faster than the LMP request handler thread. We have not understand the real reason behind. However, we do ensure that the reading is correct.

10.7 Limitations

We identify that there are the following aspects we can improve upon.

More functional. We intentionally left out some functionalities specified in POSIX. For example, in POSIX, users are allowed to set the cursor after the end of the file and therefore leave a "hole" area. Another example is the full set of flags supported by `fopen`. Some flags, e.g., `b, x`, are not taken into account during implementation.

Better performance when cache is cold. As shown in 10.6, there is a large discrepancy between the performance of cold cache and warm cache. Cold cache performance, in reality, is also important because it affects the responsiveness of the system. The stem of the problem is the sub-optimal block driver, which we tried to optimize with methods discussed in 10.5.

No support for long file names.

Chapter 11

Networking

Author: Sven Grübel

Networking is a big part in the contemporary computing world. Thus, it is intuitively clear that a modern operating system must have a strong foundation for client domains to access the internet.

Staying with the Barrelfish-theme, we decided to give our networking domain the appropriate name of "fishnet".

11.1 Architecture

In harmony with the micro-kernel spirit, networking is not implemented directly in the kernel, but rather as a separate domain. In fact, there is only one single instance of the networking domain on the whole system.

11.2 Design Choices

As the networking domain needs to interact with the hardware (specifically with the network interface card), it must have access to the device capability "cap_dev" that is controlled by the "init" domain on the bootstrap core. For simplicity's sake, the networking domain thus runs only on the bootstrap core. This way, the device capability does not have to be transferred cross-core. As an added benefit, this also assures that only one single domain accesses the hardware at any given time, and that all accesses to the hardware are sequential.

An idea that we had initially was to run a networking proxy-domain on each application core. These proxy-domains could then forward LMP requests from domains on the same core to the "real" networking domain through a dedicated direct UMP channel. However, this idea was scrapped in favor of using the standard forwarding channels in order to keep the overall system as simple as possible. Nonetheless, it is still possible for domains to create a direct UMP channel to the networking domain thanks to our versatile RPC framework. This is especially useful when lots of data is being received and/or sent (e.g. a web server).

Concerning the implementation of the whole network stack, we began by working with the already provided C header files. However, it became clear

rather quickly that they were not very useful, and we decided to create our own implementation from scratch. Here are a few reasons for this:

- The structs did not include payload fields. This is rather concerning. Did the course organizers think that we only needed the packet headers? How were we supposed to use packet payloads? In our own implementation, we use variable length fields in the network packet structs for payload fields. For IPv4 and TCP, the variable length struct field is used for both the options and the payload. Preprocessor definitions for these protocols are provided to calculate the pointer to the start of the payload based on the header fields.

- The structs were not specified as being big endian. Why would we take care of the endianness ourselves when the compiler can do it for us? To quote Prof. Roscoe form the bachelor's course *Systems Programming and Computer Architecture*: "The compiler is your friend!". This is why we use the compiler attribute "scalar_storage_order("big-endian")" to signal to the compiler that the packet structs should be interpreted as being big endian, no matter what endianness the underlying architecture has. Compilers really are our friends, aren't they?

- The structs did not use bit fields when needed. This is more for comfort and code maintainability rather than a complaint about incomplete implementation. Yes, we could use bit masks everywhere and make our code harder to read and harder to maintain - and it would still be working correctly. But we could make our lives much easier by using bit fields. All network packets use consecutive bit fields that are properly aligned for the next non-bit field, so the bit fields work correctly in the packed structs.

- The header files were public to the system. This does not make any sense. Why would we expose the network packet structs to all domains? They are only needed internally in the network domain. Other domains have no need to know what an ARP packet is, for example. Other domains should only be concerned about having a buffer of bytes and a corresponding buffer length and either sending this buffer to the network domain or receiving this buffer from the network domain. This is why we made the packet definitions local to the network domain.

Rant over.

11.3 Internal Functionality

This section describes in detail how the internal functionality of the networking domain is implemented. The networking domain handles the following internal functionality.

- Interacting with the "enet" driver
- Converting ethernet frames into packets of higher level protocols
- Converting packets of higher level protocols into ethernet frames

- Keeping an ARP cache
- Keeping a port registry

11.3.1 Interacting with the "enet" Driver

The provided "enet" driver already provides the functionality to interact with the network interface card through a device queue[1]. We only needed to map the memory-mapped registers of the network interface card into virtual memory. This was done by retyping the device capability (that is transferred from the "init" process) to a device frame with appropriate offset and length. The retyped capability was then mapped into the virtual address space and the virtual address is stored in the local driver state.

The "enet" driver actually provides two device queues, one for receiving (RX) and one for transmitting (TX). The RX queue has a total of 512 slots for received ethernet frames. This is enough for normal system operations. For the TX queue however it is enough to have a single slot for a single ethernet frame. We are not sure why a single slot suffices, but we suspect that the system is not capable of enqueueing ethernet frames faster than they are sent over the wire by the driver.

11.3.2 Receiving Network Data

Receiving network data consists of two steps: receiving raw ethernet frames from the "enet" driver and converting these ethernet frames into higher layer packets.

Receiving Ethernet Frames

Receiving raw etherent frames is done in an infinite loop in a dedicated thread. The thread is polling the RX queue of the "enet" driver. If the thread receives an error that the queue is empty, it yields in order to not waste any of the precious system time. As soon as the thread receives a buffer from the device queue, it casts the received valid data pointer to an "EthernetPacket" struct and passes it to the conversion step, together with the received valid data length of the buffer.

Converting Ethernet Frames to Higher Layer Packets

In general, converting packets from lower layer protocols to higher layer protocols is pretty much straightforward. It mostly consists of adding an offset to the payload pointer of the lower layer protocol and casting it to the next higher layer. For some protocols, the payload length needs to be passed separately, as it is not included in the packet header of some protocols.

Converting an ethernet frame to either an ARP message or an IPv4 packet consists of casting the "payload" field of the "EthernetPacket" pointer to either a "ArpPacket" pointer or a "IPv4Packet" pointer, respectively. Passing the length of the payload to the ARP handler is not necessary, as ARP messages have a fixed size of 28 bytes. Passing the length of the payload to the IPv4 handler is not

[1] See technical note 26 of Barrelfish

necessary, as the headers of the IPv4 packet hold the length in the "total_length" field.

Converting an ethernet frame into any other next layer protocol is currently not supported and will result in the received ethernet frame being dropped silently.

Converting an IPv4 packet to either an ICMP message, a UDP datagram, or a TCP segment is also straightforward. In all three cases, the payload base address of the "IPv4Packet" struct is cast to the struct of the corresponding protocol. The payload base address can be calculated directly from the IPv4 packet by adding the options length as offset to the "options_and_payload" field of the "IPv4Packet" struct. The options length can be calculated by subtracting 20 bytes (the size of the fixed-size header fields) from the total header length which is stored in the "ihl" field as number of 32-bit words. As a side note: the source IPv4 address is passed to the next layer in all three cases. This way, end-clients can respond to requests. Most application layer protocols do not track the source IPv4 address, and without passing it, responding to network requests would be mostly unachievable.

Converting an IPv4 packet into any other next layer protocol is currently not supported and will result in the received IPv4 packet being dropped silently.

11.3.3 Sending Network Data

In order to send network data, the same two steps need to be performed as for receiving network data, except of course in the reverse order.

Converting Higher Layer Packets to Ethernet Frames

Generally speaking, converting a higher layer packet to the next lower layer packet usually consists of the following steps:

- Allocating a buffer that is large enough to hold the lower layer packet headers plus the whole upper layer packet
- Setting the correct header fields based on additional input
- Copying the higher layer packet into the payload field of the lower layer packet
- (If applicable) Calculating the internet checksum over the complete lower layer packet, sometimes including a prepended pseudo-header, and writing it to the appropriate struct field of the lower layer packet

A question that might arise immediately is why the packet is copied every time there is a switch in protocol layers. That is indeed a downside of always wanting to keep the packets in one continuous block of memory. However, we believe that the benefits of always having the packet ready in one place outweigh the downside of a not-so-memory-heavy implementation. For example, the calculation of the internet checksum is trivial when the packet is in a continuous block of memory. Additionally, when talking about performance, it is important to keep in mind that networking usually sees natural delays in the order of milliseconds. So from the networking perspective, the memory copying overhead is negligible.

Starting from the bottom of the supported protocols: Converting an ICMP message, a UDP datagrams, or a TCP segments into an IPv4 packet is achieved by additionally passing the destination IPv4 address. Internally, the packet headers are then set accordingly, with the TOS, identification, DF flag, MF flag, and fragment offset all set to zero, as these fields provide features that are not needed in this implementation. The options list is also kept empty for the same reason. The TTL field of the IPv4 header is initially set to 64, as recommended by IANA[2].

In order to convert an IPv4 packet to an ethernet frame, the destination MAC address of the machine that has the destination IPv4 address must be determined. This is done by querying the ARP cache (described in more detail in Section 11.3.4). If the ARP cache does not find a corresponding MAC address, an error is returned. If the ARP cache finds a corresponding MAC address, it is used as the destination MAC address in the ethernet frame.

The conversion of ARP messages to ethernet frames is a bit simpler. Either the ARP message is a request or it is a response. If it is a request, the destination MAC address is simply the well-known broadcast MAC address. And if the ARP message is a response, the destination MAC address is known from the corresponding request that triggered the generation of the response.

As a quick summary, to create an ethernet frame from a UDP datagram, all that is additionally needed is an IP address. The ARP cache will take care of figuring out the corresponding MAC address such that an ethernet frame for the correct destination can be created.

Sending Ethernet Frames

All that needs to be done for sending raw ethernet frames is to enqueue the frame to the transmission queue of the "enet" driver.

11.3.4 Keeping an ARP Cache

For the conversion of IPv4 packets to ethernet frames, the network domain needs a way to find out the destination MAC address that corresponds to the destination IPv4 address of the IPv4 packet. For this process, the *Address Resolution Protocol* (ARP) is used. ARP is described in detail in RFC 826[3].

For the implementation of the ARP cache, the lookup table is implemented as a hash table. The hash table is indexed by the IPv4 address and additionally stores the corresponding MAC address. The RFC specifies that the protocol type should be saved as well. However, as we only support IPv4 as protocol type, this is not needed for this implementation.

The implementation supports thread safety, so multiple threads can simultaneously query the ARP cache or update entries in it without seeing any inconsistent state. This is done by acquiring a lock before each operation on the hash table, and releasing the lock after finishing the operation. The lock is marked as static in the ARP translation unit, so it is not accessible from outside.

On arrival of any ARP message, it is checked if the ARP cache already holds an entry with the sender protocol address as key. If it does, the entry is updated with the sender hardware address as the value. If it does not, an entry

[2]https://www.iana.org/assignments/ip-parameters/ip-parameters.xml
[3]https://datatracker.ietf.org/doc/html/rfc826

is generated with the sender protocol address as key and the sender hardware address as value. After the insertion or update, the operation code of the ARP message is checked. If the operation is a request and the target physical address corresponds to the IPv4 address of the machine, a reply is generated to inform the sender that the target protocol address is operated by the hardware address of the network interface card of the machine.

As described in Section 11.3.3, the machine performs an ARP lookup to translate an IPv4 address to the corresponding MAC address. This is done by querying the hash table for an entry that has the IPv4 address as key. If such an entry exists, the corresponding MAC address is returned.

If there is no entry for the IPv4 address, the machine generates a new ARP query and broadcasts it over the network by using the well-known broadcast MAC address `FF:FF:FF:FF:FF:FF` as destination. If the query is answered, the reply is handled as described in a previous paragraph.

Note that the reply is handled by a different thread than the one that is querying the ARP cache. Because of this, the querying thread does not know when (or even if) a reply arrives. To solve this issue, the querying thread sleeps for a certain amount of time. After waking up, the querying thread checks the ARP cache again. If there is no entry yet, a new ARP request is broadcast (NB: the rebroadcasting is needed because the ARP query might have been lost in the network, as sending network data is in general unreliable). The procedure is repeated for a certain number of rounds. If after the specified number of rounds there is still no corresponding entry in the cache, the IPv4 address is deemed to be unreachable and an error is returned to the client that tried sending the initial IPv4 packet.

During the implementation, the following parameters were decided upon:

- Thread sleep: 50ms
- Number of rounds: 3

The sleeping time of 50ms was deemed large enough for local network delays. This time window also never posed any problems during testing. As for the number of rounds, we believe that 3 rounds give a rather high assurance that a host is not reachable, as 3 network messages being dropped consecutively is very unlikely.

11.3.5 Keeping a Port Registry

The port registry is responsible for keeping track of all open ports and which domains are listening on these open ports. To achieve this functionality, the port registry holds a local lookup table that maps a port number to a <core, domain>-tuple. Additionally, each entry stores whether the client is listening for UDP data, TCP data, or both. The lookup table is implemented as a hash table with the port number being the key.

When a client requests to listen to a port, it is first checked whether the port is already registered. If it is, an error is returned to the client. If not, a new entry is inserted into the lookup table, storing the client's core ID and domain ID, together with the requested listening protocols (UDP, TCP, or both). Additionally, a direct RPC channel from the network domain to the listening

client is created for reduced latency. Note that the current implementation has no support for TCP.

When some network data arrives over UDP, the port registry is asked by the UDP message handler to forward the data to the <core, domain>-tuple that is listening on the destination port. If there is no entry in the lookup table for the destination port, the port registry returns an error to the UDP message handler and the UDP datagram is silently dropped. On the other hand, if the port registry finds an entry in its lookup table for the port, it is checked whether the client wishes to listen to UDP data. If this is not the case, the port registry again returns an error to the UDP message handler and the UDP datagram is silently dropped. If the client wishes to listen to UDP data on the destination port, the port registry sends the UDP payload buffer to the listening client over the RPC framework by using the direct RPC channel that was established when requesting to listen to the port. Some additional metadata is also sent to the client, such as the destination port, the source IPv4 address, the source port, as well as whether the data was received over UDP or TCP.

11.4 External Functionality

This section describes in detail how the functionality of the networking domain is implemented. The networking domain provides the following three functionalities to all other domains.

- Listening on a Port
- Receiving Network Data
- Sending Network Data

The following subsections provide detailed information about each of these external functionalities.

11.4.1 Listening to a Port

Client domains may request to be notified about arriving network data on a specific destination port. This process is called *listening to a port*. A domain must register itself in the port registry (see Section 11.3.5) in order for it to receive the network data that arrives at this port. Clients may interact with the port registry through the following RPC function:

```
1  /*
2   * aos_rpc_network_listen - Listen to a port for network data
3   *
4   * @param port       - The port to listen on
5   * @param listen_udp - If the domain should listen for UDP data
6   * @param listen_tcp - If the domain should listen for TCP data
7   *
8   * @return SYS_ERR_OK on success, error code otherwise
9   */
10 errval_t aos_rpc_network_listen(uint16_t port, bool listen_udp,
       bool listen_tcp);
```

The function sends an RPC message of type `NETWORK_PORT_REQUEST` to the networking domain. The networking domain then handles this message as described in Section 11.3.5. If the port was successfully registered, the RPC call returns `SYS_ERR_OK`, otherwise an appropriate error code is returned.

During the handling of the port registration message, the networking domain will establish a direct RPC channel to the client domain. This helps with drastically reducing the latency for the very first network data packet that is received for this port, as otherwise the direct channel will only be set up on receiving the first network data packet.

11.4.2 Receiving Data

(Note that TCP is not supported in the current implementation. This section thus only describes how receiving UDP data works.)

When the UDP handler receives a UDP datagram from the network (see Section 11.3.2), it is transferred to the port registry for system-wide distribution. As described in Section 11.3.5, the port registry sends the received network data together with some necessary metadata to the listening client domain through the previously established direct channel. It is again important to note that this channel is guaranteed to exist, as the port registration would not have succeeded if the channel setup failed in any way.

The RPC message that is sent by the networking domain is of the type `NETWORK_DATA_REQUEST`. Clients receive this message in their RPC message handler. Thus, clients should implement their own RPC message handler function and set it up by using the `rpc_start_server_request_handling` function, as otherwise the standard RPC request handler will reject the message.

Clients should immediately copy the network data to a local buffer and reply to the RPC request, as the packet handling thread of the networking domain is blocked until the reply arrives. Clients can then process the network data in a separate thread. The "echo" domain gives an example on how this is achieved.

11.4.3 Sending Data

Sending data over the network is very straightforward. The following RPC function is used for this purpose:

```
/*
 * aos_rpc_network_send - Send data over the network
 *
 * @param src_port - From which port the data should be sent
 * @param dst_port - To which remote port to send the data
 * @param dst_ipv4 - To which remove IPv4 address to send the data
 * @param is_udp   - Flag to send over UDP or TCP
 * @param buf      - The base pointer to the buffer to be sent
 * @param buf_len  - The length of the buffer to be sent
 *
 * @return SYS_ERR_OK on success, error code otherwise
 */
errval_t aos_rpc_network_send(uint16_t src_port, uint16_t dst_port,
     uint32_t dst_ipv4, bool is_udp, void *buf, size_t buf_len);
```

Note that sending data over TCP is not supported in the current implementation, thus setting the `is_udp` flag to `false` will always result in an error.

11.5 Putting it Together

Figure 11.1 gives an overview how receiving data works.

Generally speaking, the networking domain receives raw ethernet frames from the receive queue of the "enet" driver. These raw ethernet frames are then transformed into their innermost protocol. ARP and ICMP messages are handled internally inside the networking domain. UDP datagrams are reported to the listening client domain, if it exists. TCP segments are silently dropped, as the current implementation does not support it.

For client domains to be able to receive network data, they must request to listen to a port in the port registry of the networking domain.

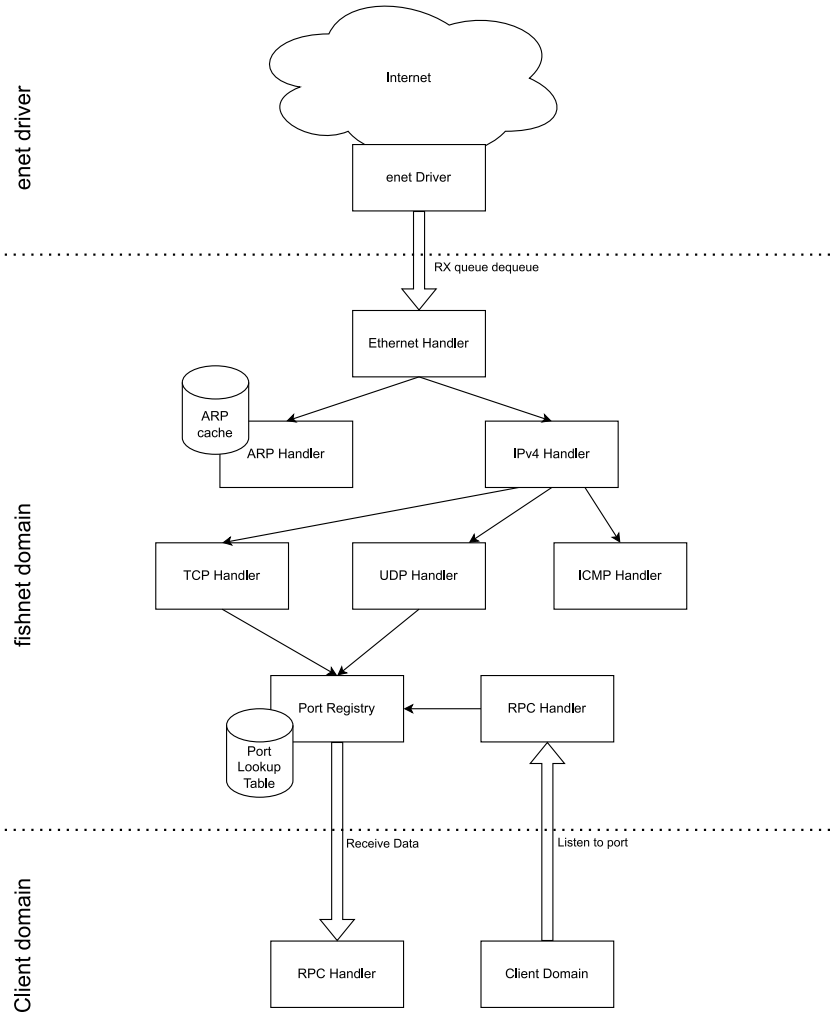


Figure 11.1: Overview of receiving network data

Figure 11.2 gives an overview how sending data works. Client domains can choose between sending data fast through UDP, or reliably through TCP. Note that TCP currently is not implemented and will return an error to the client

domain.

The network data to be sent is received in the RPC handler of the networking domain. An appropriate transport layer packet is generated which is then transformed down the layers until it is a raw ethernet frame.

If the ARP lookup (see Section 11.3.4) fails for whatever reason, this error is cascaded back to the client that requested the network data sending. If there is any other failure, the error is also cascaded back to the client.

If all works well, the UDP packet is transformed all the way down the networking stack into an ethernet frame. This ethernet frame is then enqueued into the transmit queue of the "enet" driver which takes care of sending the ethernet frame on the wire.

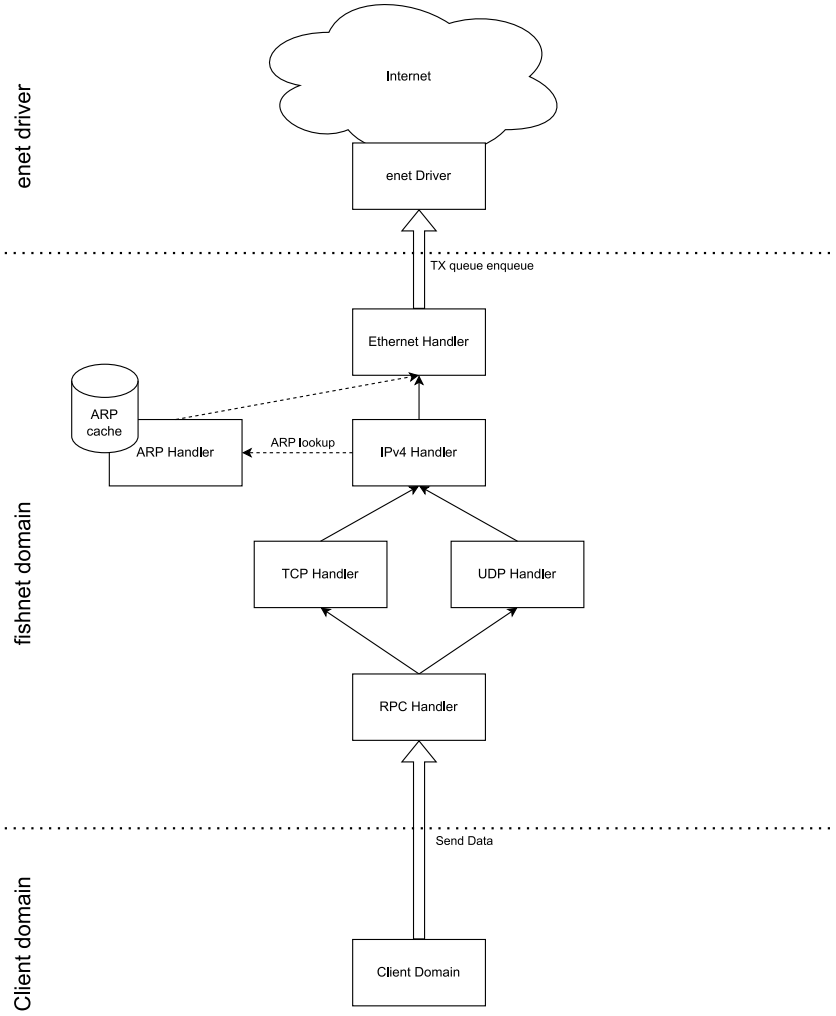


Figure 11.2: Overview of sending network data

11.6 Performance Evaluation

The following performance evaluations have been performed by connecting the machine that is running the OS to another host using an ethernet cable. The connection was configured to 100 MBit/s. It is important to note that these measurements were performed with debug printing disabled.

Latency measurement using ICMP . The external host used the "ping" utility to measure the latency to the machine. "ping" uses ICMP messages for its statistics calculations. ICMP messages are handled only inside the networking domain, so there is no RPC messaging overhead. The "ping" utility repeatedly reported back a round-trip latency of **less than 1 millisecond**.

Latency measurement using UDP . The machine started an echo server on the standard port 7. The external host sent small UDP packets one-by-one to this port and recorded the time until a response was received for each packet. The average round-trip latency for the first message was measured to be **around 1 millisecond**.

Throughput measurement using UDP . The machine started an echo server on the standard port 7. The external host sent many large UDP packets at once to this port and recorded the total time until a response was received for all packets. The number of transmitted bits was determined by also taking the ethernet header and the UDP header size into account. Peak performance was achieved **around 68.9 MBit/s**.

11.7 Limitations

Of course, no system is endlessly brilliant. The following list shows some of the limitations of our networking implementation.

- TCP is currently not supported.
- IPv4 fragmentation is currently not supported. This is mostly because it is specifically listed as not required in the project description.
- IPv4 address conflict detection is not implemented, so the IPv4 address of the machine must be set very carefully. IPv4 address conflict detection is described in RFC 5227[4].
- Deregistering a port is currently not possible. The functionality is actually implemented in the port registry. However, as no teardown is implemented for RPC channels, the functionality is not exposed to other domains through an RPC function.

[4]https://datatracker.ietf.org/doc/html/rfc5227

Chapter 12

Conclusions

We had now arrived at an end. Looking back, we are satisfied with the final product and in general with our design decision. We are well aware that time is a scarce resource in a project like this and that being able to balance the ambitions with the ability to actually develop the ideas themselves is part of the challenge. Thus, we tried out best to settle on as less compromises as possible but we are aware that some limitations are still in place as almost every chapter has pointed out.

Overall, we all learnt **a lot**.

12.1 Notes about Grading

We noticed shortly before the deadline that the grading function "grading_test_mm()" was called in the wrong place (namely on line 716 of usr/init/main.c). We moved the function call to the correct position, but we did not have time to test it. The previous function call is in place on the mentioned line, but it is currently commented out.

12.2 End of life

Today, the 2nd of June 2022.

After all, we don't want anyone to expect security updates...

Appendix A

TLA+ Pretty Printed Specification

MODULE *capsystem*

EXTENDS *Sequences*, *Naturals*, *FiniteSets*

CONSTANTS *Cores*, *BaseCaps*

VARIABLES *localCaps*, *messageQueues*, *deleteHandlers*, *requestHandlers*

CONSTANT DEFINITIONS (IN *TLC*) (Both as Ordinary *Assinments*)

```
Cores ← {"C1", "C2"}
BaseCaps ← [C1 ↦
{
  [owner ↦ "C1",
  remote_copies ↦ FALSE,
  remote_ancs ↦ FALSE,
  remote_descs ↦ FALSE,
  locked ↦ FALSE,
  in_delete ↦ FALSE,
  start ↦ 0,
  end ↦ 2]
},
C2 ↦ {}
]
```

BEHAVIOUR SPEC USED IN *TLC*

$\wedge$ *CapSystemInit*
$\wedge \Box[CapSystemStep]\_\langle localCaps, messageQueues, requestHandlers, deleteHandlers\rangle$
$\wedge \forall\, c \in Cores$: WF_$\langle localCaps, messageQueues, requestHandlers, deleteHandlers\rangle(ReceiveMsg(c))$

$\wedge \forall\, c \in Cores$: WF_$\langle localCaps, messageQueues, requestHandlers, deleteHandlers\rangle(ProcessRequest(c))$

$\wedge \forall\, c \in Cores$: WF_$\langle localCaps, messageQueues, requestHandlers, deleteHandlers\rangle(DeleteStep(c))$

$\wedge \forall\, c \in Cores$: WF_$\langle localCaps, messageQueues, requestHandlers, deleteHandlers\rangle(DeleteComplete(c))$

CHECKED INVARIANTS (IN *TLC*)

CapSystemTypeOK
CapSystemInvariants
RevokeInvariants

CHECKED LIVENESS PROPERTY (IN *TLC*)

$\forall\, c \in Cores$: $\Box\Diamond(requestHandlers[c].state = \text{"IDLE"})$

TYPE DEFINITIONS

$$
\begin{array}{l}
\mathit{Capabilities} \triangleq \\
[\\
\mathit{owner} : \mathit{Cores}, \\
\mathit{remote\_copies} : \text{BOOLEAN} , \\
\mathit{remote\_descs} : \text{BOOLEAN} , \\
\mathit{remote\_ancs} : \text{BOOLEAN} , \\
\mathit{locked} : \text{BOOLEAN} , \\
\mathit{in\_delete} : \text{BOOLEAN} , \\
\mathit{start} : \mathit{Nat}, \\
\mathit{end} : \mathit{Nat} \\
]
\end{array}
$$

$$\mathit{Range}(f) \triangleq \{f[x] : x \in \text{DOMAIN } f\}$$

$$
\begin{array}{l}
\text{RECURSIVE } \mathit{SeqFromSet}(\_) \\
\mathit{SeqFromSet}(S) \triangleq \\
\quad \text{IF } S = \{\} \text{ THEN } \langle\rangle \\
\quad\quad \text{ELSE LET } x \triangleq \text{CHOOSE } x \in S : \text{TRUE} \\
\quad\quad\quad\quad \text{IN } \langle x \rangle \circ \mathit{SeqFromSet}(S \setminus \{x\})
\end{array}
$$

$$
\begin{array}{l}
\mathit{SendCapMsg} \triangleq [\mathit{type} : \{\text{“SendCapMsg”}\}, \\
\qquad\qquad\qquad\quad \mathit{cap} : \mathit{Capabilities}, \\
\qquad\qquad\qquad\quad \mathit{sender} : \mathit{Cores}]
\end{array}
$$

$$
\begin{array}{l}
\mathit{RevokeMark} \triangleq [\mathit{type} : \{\text{“RevokeMark”}\}, \\
\qquad\qquad\qquad\quad \mathit{cap} : \mathit{Capabilities}, \\
\qquad\qquad\qquad\quad \mathit{sender} : \mathit{Cores}]
\end{array}
$$

$$
\begin{array}{l}
\mathit{RevokeMarkDone} \triangleq [\mathit{type} : \{\text{“RevokeMarkDone”}\}, \\
\qquad\qquad\qquad\qquad\quad \mathit{sender} : \mathit{Cores}]
\end{array}
$$

$$
\begin{array}{l}
\mathit{RevokeSweep} \triangleq [\mathit{type} : \{\text{“RevokeSweep”}\}, \\
\qquad\qquad\qquad\quad \mathit{sender} : \mathit{Cores}]
\end{array}
$$

$$
\begin{array}{l}
\mathit{RevokeSweepDone} \triangleq [\mathit{type} : \{\text{“RevokeSweepDone”}\}, \\
\qquad\qquad\qquad\qquad\quad \mathit{sender} : \mathit{Cores}]
\end{array}
$$

$$
\begin{array}{l}
\mathit{DeletionNotice} \triangleq [\mathit{type} : \{\text{“DeletionNotice”}\}, \\
\qquad\qquad\qquad\qquad\quad \mathit{cap} : \mathit{Capabilities}, \\
\qquad\qquad\qquad\qquad\quad \mathit{sender} : \mathit{Cores}]
\end{array}
$$

$$
\begin{array}{l}
\mathit{DeletionAck} \triangleq [\mathit{type} : \{\text{“DeletionAck”}\}, \\
\qquad\qquad\qquad\quad \mathit{sender} : \mathit{Cores}, \\
\qquad\qquad\qquad\quad \mathit{transfer} : \text{BOOLEAN} , \\
\qquad\qquad\qquad\quad \mathit{cap} : \mathit{Capabilities}]
\end{array}
$$

$$
\begin{aligned}
CanRetypeMsg \triangleq\ & [type : \{\text{“CanRetypeMsg”}\},\\
& sender : Cores,\\
& cap : Capabilities,\\
& retype\_cap : Capabilities]
\end{aligned}
$$

$$
\begin{aligned}
CanRetypeAnswer \triangleq\ & [type : \{\text{“CanRetypeAnswer”}\},\\
& sender\ : Cores,\\
& answer : \text{BOOLEAN}\ ,\\
& transfer : \text{BOOLEAN}\ ]
\end{aligned}
$$

$$
\begin{aligned}
RetypeTransferAck \triangleq\ & [type : \{\text{“RetypeTransferAck”}\},\\
& sender\ : Cores,\\
& answer : \text{BOOLEAN}\ ,\\
& cap : Capabilities]
\end{aligned}
$$

$$
\begin{aligned}
OwnershipTransferMsg \triangleq\ & [type : \{\text{“OwnershipTransfer”}\},\\
& sender : Cores,\\
& cap : Capabilities]
\end{aligned}
$$

$$
\begin{aligned}
OwnershipTransferAck \triangleq\ & [type\ \ : \{\text{“OwnershipTransferAck”}\},\\
& sender : Cores,\\
& cap : Capabilities]
\end{aligned}
$$

$$
\begin{aligned}
Messages \triangleq\ & SendCapMsg\\
\cup\ & RevokeMark\\
\cup\ & RevokeMarkDone\\
\cup\ & RevokeSweep\\
\cup\ & RevokeSweepDone\\
\cup\ & DeletionNotice\\
\cup\ & DeletionAck\\
\cup\ & CanRetypeMsg\\
\cup\ & CanRetypeAnswer\\
\cup\ & RetypeTransferAck\\
\cup\ & OwnershipTransferMsg\\
\cup\ & OwnershipTransferAck
\end{aligned}
$$

$$
\begin{aligned}
DeleteState \triangleq\ & \{\text{“COORDINATING”},\ \text{“READY”}\}\\
RequestState \triangleq\ & \{\text{“TRANSFER”},\\
& \text{“DELETE\_PROCESSING”},\\
& \text{“REVOKE\_MARK”},\\
& \text{“REVOKE\_SWEEP”},\\
& \text{“RETYPE\_FOR\_REMOTE”},\\
& \text{“RETYPE\_QUERYING”},\\
& \text{“IDLE”}\}
\end{aligned}
$$

$DeleteHandler \triangleq$
[
state of deletion handler
$state : DeleteState,$
how many times deletion handler needs to be resumed to be allowed to continue
$pending\_resumes : Nat,$
capability for which deletion/transfer is coordinated
$cap : Capabilities \cup \{[owner \mapsto \text{“None”}]\},$
$inform\_request\_handler : \text{BOOLEAN}\ ,$
msgs to send when done
$msgs : \text{SUBSET}\ Messages$
]

$RequestHandler \triangleq$
[
$state : RequestState,$
$cap : Capabilities \cup \{[owner \mapsto \text{“None”}]\},$
$retype\_cap : Capabilities \cup \{[owner \mapsto \text{“None”}]\},$
$delete\_done : \text{BOOLEAN}\ ,$
$ack\_received : \text{BOOLEAN}$
]

CAPABILITY HELPER OPERATORS

$$CapContained(child,\ parent) \triangleq \begin{array}{l} \land\ child \in Capabilities \\ \land\ parent \in Capabilities \\ \land\ parent.start \leq child.start \\ \land\ child.end \leq parent.end \end{array}$$

$$HasAncs(cap,\ capSet) \triangleq \exists\, cap2 \in capSet : CapContained(cap,\ cap2) \land \neg CapContained(cap2,\ cap)$$

returns the set of smallest common ancestors of child in *capSet*

$$\begin{array}{l} getParent(capSet,\ child) \triangleq \\ \quad \{c \in capSet : \begin{array}{l} \land\ CapContained(child,\ c) \\ \land \neg CapContained(c,\ child) \\ \land \neg(\exists\, c2 \in capSet \setminus \{c\} : \begin{array}{l} \land \neg CapContained(c2,\ child) \\ \land\ CapContained(child,\ c2) \\ \land\ CapContained(c2,\ c))\} \end{array} \end{array} \end{array}$$

$$HasLocalDescs(cap,\ capSet) \triangleq \exists\, cap2 \in capSet : \begin{array}{l} \land\ getParent(capSet,\ cap2) = \{cap\} \\ \land\ cap2.owner = cap.owner \end{array}$$

$$CapIsCopy(c1,\ c2) \triangleq c1.start = c2.start \land c1.end = c2.end$$

$CapsDisjoint(c1,\ c2) \triangleq \quad \lor\ c1.end \leq c2.start$
$\lor\ c2.end \leq c1.start$

INITIALIZATION

$CapSystemInit \triangleq$
$\land\ localCaps = [c \in Cores \mapsto BaseCaps[c]]$
$\land\ messageQueues = [c \in Cores \mapsto \langle\rangle]$
$\land\ deleteHandlers = [c \in Cores \mapsto [$
$state \mapsto$ "READY",
$pending\_resumes \mapsto 0,$
$cap \mapsto [owner \mapsto$ "None"$],$
$inform\_request\_handler \mapsto$ FALSE,
$msgs \mapsto \{\}$
$]]$
$\land\ requestHandlers = [c \in Cores \mapsto [$
$state \mapsto$ "IDLE",
$cap \mapsto [owner \mapsto$ "None"$],$
$retype\_cap \mapsto [owner \mapsto$ "None"$],$
$delete\_done \mapsto$ FALSE,
$ack\_received \mapsto$ FALSE
$]]$

INVARIANTS

$CapSystemTypeOk \triangleq$
$\land\ localCaps \in [Cores \to$ SUBSET $Capabilities]$
$\land\ messageQueues \in [Cores \to Seq(Messages)]$
$\land\ \forall\ c \in Cores : (\forall\ cap \in localCaps[c] :$
$\exists\ cap2 \in$ UNION $\{BaseCaps[core] : core \in Cores\} :$
$\land\ CapContained(cap,\ cap2)$
$\land\ cap.start < cap.end)$
$\land\ deleteHandlers \in [Cores \to DeleteHandler]$
$\land\ requestHandlers \in [Cores \to RequestHandler]$
at most one message to send when delete queue finishes
$\land\ \forall\ c \in Cores : Cardinality(deleteHandlers[c].msgs) \leq 1$
ensure message queue is bounded
$\land\ \forall\ c \in Cores : Len(messageQueues[c]) < 4$

$RevokeInvariants \triangleq$
when we have sent the "RevokeSweep" message, all descendants of the target must be marked for deletion and the target must be locked
$\forall\ c \in Cores :$
LET $rh \triangleq requestHandlers[c]$IN
$(\exists\ c2 \in Cores \setminus \{c\} :$

$\land\ messageQueues[c2] \neq \langle\rangle$
$\land\ Head(messageQueues[c2]).type = \text{“RevokeSweep”})$
$\Rightarrow$
(
all strict descendants are in delete on every core
$\land \neg\exists\ cap \in \text{UNION}\ \{localCaps[core] : core \in Cores\} :$
$(\land\ CapContained(cap,\ rh.cap)$
$\land \neg CapContained(rh.cap,\ cap)$
$\land \neg cap.in\_delete)$
no copies exist on any core but c
$\land \neg\exists\ cap \in \text{UNION}\ \{localCaps[core] : core \in Cores \setminus \{c\}\} : CapIsCopy(cap,\ rh.cap)$
a copy exists on core c and it's *locked* –
note that we only check for “a” locked copy, which sufficies because of the copy uniqueness invariant below
$\land\ \exists\ cap \in localCaps[c] : CapIsCopy(cap,\ rh.cap) \land cap.locked$
there are not messages with descendants or copies in-flight -
because that would mean they survive the revocation!
$\land \neg\exists\ core \in Cores : \exists\ msg \in Range(messageQueues[core]) :$
$(\land (msg.type = \text{“OwnershipTransfer”} \lor msg.type = \text{“SendCapMsg”})$
$\land\ CapContained(msg.cap,\ rh.cap))$
)

$CapSystemInvariants \triangleq$
caps are non-overlapping or contained
$\land \forall\ cap1,\ cap2 \in \text{UNION}\ \{localCaps[c] : c \in Cores\} : \lor\ CapsDisjoint(cap1,\ cap2)$
$\lor\ CapContained(cap1,\ cap2)$
$\lor\ CapContained(cap2,\ cap1)$
remote relations of all copies agree at each core, *i.e.* there is only exactly one cap copy in the set
$\land \forall\ c \in Cores : \forall\ cap \in localCaps[c] : \neg\exists\ cap2 \in localCaps[c] : cap \neq cap2 \land CapIsCopy(cap,\ cap2)$
owner has a copy (unless it is marked as “*in_delete*” everywhere else), potentially with differing remote relations (!)
$\land \forall\ cap \in \text{UNION}\ \{localCaps[c] : c \in Cores\} : \lor\ cap.in\_delete$
$\lor (\exists\ cap2 \in localCaps[cap.owner] :$
$\land\ cap2.owner = cap.owner$
$\land\ cap2.start\ \ = cap.start$
$\land\ cap2.end = cap.end)$
if the same capability exists on two different cores:
$\land \forall\ c1 \in Cores : \forall\ c2 \in Cores \setminus \{c1\} :$
$\forall\ cap1 \in localCaps[c1],\ cap2 \in localCaps[c2] :$
$CapIsCopy(cap1,\ cap2) \Rightarrow$
either the owners agree and remote copy flags are set unless the copy is in delete (revocation marking unsets remote copy flag)
$\lor\ \land\ cap1.owner = cap2.owner$
$\land\ (cap1.remote\_copies \lor cap1.in\_delete)$
$\land\ (cap2.remote\_copies \lor cap2.in\_delete)$

or an ownership transfer of the capability is in progress that explains the ownership mismatch

$$\vee \exists\, core \in \{cap1.owner,\ cap2.owner\} : \exists\, msg \in Range(messageQueues[core]) :$$
$$(\ \wedge\ (\ \vee\ (msg.type = \text{“DeletionAck”} \wedge msg.transfer)$$
$$\vee\ msg.type = \text{“OwnershipTransferAck”}$$
$$\vee\ (msg.type = \text{“RetypeTransferAck”} \wedge msg.answer))$$
$$\wedge\ CapIsCopy(msg.cap,\ cap1))$$

remote descendants (unless revocation in progress $\rightarrow$ removes *remote_descs*) are correctly set:
for all child-parent pairs of capabilities that have copies with different owners:

$$\wedge\ \ \forall\, c \in Cores : \forall\, child \in localCaps[c] :$$
$$\forall\, parent \in getParent(\text{UNION}\ \{localCaps[core] : core \in Cores\},\ child) :$$
$$child.owner \neq parent.owner \Rightarrow$$
$$\forall\, parent\_owner\_cap \in localCaps[parent.owner] :$$
$$\vee\ \neg CapIsCopy(parent,\ parent\_owner\_cap)$$

if *parent_owner_cap* is the owner's version of the parent cap:
either remote descendants is set $\rightarrow$ because of cap that is owned by another core

$$\vee\ parent\_owner\_cap.remote\_descs$$

or there is a copy of cap at the core owning of the parent
with remote copies set $\rightarrow$ that copy tracks the existence of child

$$\vee\ (\exists\, childcopy \in localCaps[parent.owner] :$$
$$\wedge\ CapIsCopy(childcopy,\ child)$$
$$\wedge\ childcopy.remote\_copies)$$

child is being revoked, that is why the parent doesn't track its existence

$$\vee\ child.in\_delete$$

or there is an on-going ownership change for the child that explains the discrepancy
note that child that is being transferred must have been originally owned by the *parent.owner* core
if this were not the case a *childcopy* would exist / remote descs of parent would be set

$$\vee\ (\exists\, msg \in Range(messageQueues[parent.owner]) :$$
$$\wedge\ (\ \vee\ (msg.type = \text{“DeletionAck”} \wedge msg.transfer)$$
$$\vee\ msg.type = \text{“OwnershipTransferAck”}$$
$$\vee\ (msg.type = \text{“RetypeTransferAck”} \wedge msg.answer))$$
$$\wedge\ CapIsCopy(msg.cap,\ child))$$

or there is an on-going ownership change for the parent that explains the discrepancy
$\Rightarrow$ new owner already retyped locally

$$\vee\ (\exists\, msg \in Range(messageQueues[parent.owner]) :$$
$$(\ \vee\ (msg.type = \text{“DeletionAck”} \wedge msg.transfer)$$
$$\vee\ msg.type = \text{“OwnershipTransferAck”}$$
$$\vee\ (msg.type = \text{“RetypeTransferAck”} \wedge msg.answer))$$
$$\wedge\ CapIsCopy(msg.cap,\ parent))$$

remotely-owned capabilities are never locked or in deletion

$$\wedge\ \forall\, c \in Cores : \forall\, cap \in localCaps[c] : cap.owner \neq c \Rightarrow (\neg cap.locked \wedge \neg cap.in\_delete)$$

MESSAGE HELPER OPERATORS

$$SendMessage(msg,\ core) \triangleq\ messageQueues' = [messageQueues \text{ EXCEPT } ![core] = Append(messageQueues[core],\ msg)]$$

$$DequeueMsg(c) \triangleq\ messageQueues' = [messageQueues \text{ EXCEPT } ![c] = Tail(messageQueues[c])]$$

$$Broadcast(msg,\ sender) \triangleq\ messageQueues' = [c \in Cores \mapsto \text{IF } c = sender \text{ THEN } messageQueues[c] \text{ ELSE } Append(messageQueues[c],\ msg)]$$

$$BroadcastDequeue(msg,\ sender) \triangleq\ messageQueues' = [c \in Cores \mapsto \text{IF } c = sender \text{ THEN } Tail(messageQueues[c]) \text{ ELSE } Append(messageQueues[c],\ msg)]$$

EXECUTION STEP DEFINITIONS

proposed retype cap is either disjoint from or strictly contained (we don't model copies) in each other present cap

$$CanRetype(cap,\ retype\_cap,\ c) \triangleq\ \forall\, cap2 \in localCaps[c] :\ \land (\lor cap2.end \leq retype\_cap.start \lor retype\_cap.end \leq cap2.start \lor CapContained(cap,\ cap2)) \land (retype\_cap.end \neq cap2.end \lor retype\_cap.start \neq cap2.start)$$

$$LocalRetype(c,\ cap,\ retype\_cap) \triangleq$$

cap is not in the process of being deleted

$$\land \neg cap.in\_delete$$

cap is not locked

$$\land \neg cap.locked \land localCaps' = [localCaps \text{ EXCEPT } ![c] = localCaps[c] \cup \{retype\_cap\}] \land \text{UNCHANGED } (\langle messageQueues,\ requestHandlers,\ deleteHandlers\rangle)$$

$$SimpleDelete(cap,\ c) \triangleq\ localCaps' = [localCaps \text{ EXCEPT } ![c] =$$

local caps but with cap removed

$$(localCaps[c] \setminus (\{copy \in localCaps[c] : CapIsCopy(cap,\ copy)\}$$

and with the current parent version removed

$$\cup\, getParent(localCaps[c],\ cap)))$$

and with the parent with updated remote descendants flag added

$$\cup \{[parent \text{ EXCEPT } !.remote\_descs = \lor parent.remote\_descs \lor cap.remote\_descs \lor cap.remote\_copies] : parent \in getParent(localCaps[c],\ cap)\}]$$

$LocalDelete(c) \triangleq \exists\, cap \in localCaps[c] :$
$\quad \land \lor cap.owner \neq c$
no remote copies because that requires coordination
$\quad\quad \lor \land \neg cap.remote\_copies$
ensure that root caps are not deleted, otherwise we eventually end up with no capabilities
$\quad\quad\quad \land HasAncs(cap,\ localCaps[c])$
$\quad\quad\quad \land \neg cap.locked$
$\quad\quad\quad \land \neg cap.in\_delete$
$\quad \land SimpleDelete(cap,\ c)$
$\quad \land \text{UNCHANGED } (\langle messageQueues,\ deleteHandlers,\ requestHandlers\rangle)$

$SendCap(c) \triangleq \exists\, cap \in localCaps[c] :$
$\quad \exists\, c2 \in Cores \setminus \{c\} :$
$\quad \land requestHandlers[c].state = \text{“IDLE”}$
ensure that queue length is bounded
$\quad \land Len(messageQueues[c2]) < 1$
may not send a capability that is locked
$\quad \land \neg cap.locked$
may not send a capability that is about to be deleted
$\quad \land \neg cap.in\_delete$
$\quad \land localCaps' = [localCaps \text{ EXCEPT}$
$\quad\quad\quad\quad ![c] = (localCaps[c] \setminus \{cap\})$
$\quad\quad\quad\quad\quad \cup \{[cap \text{ EXCEPT}$
$\quad\quad\quad\quad\quad\quad !.remote\_copies = \text{TRUE}]\}]$
$\quad \land Broadcast([type \mapsto \text{“SendCapMsg”},\ cap \mapsto cap,\ sender \mapsto c],\ c)$
$\quad \land \text{UNCHANGED } (\langle deleteHandlers,\ requestHandlers\rangle)$

If we already have a copy with matching *owner* $\rightarrow$ inherit existing remote relations
$ReceiveCapMsg(c) \triangleq$
$\quad \land (\text{CASE } (\exists\, cap \in localCaps[c] :$
$\quad\quad \land CapIsCopy(cap,\ Head(messageQueues[c]).cap)$
$\quad\quad \land cap.owner = Head(messageQueues[c]).cap.owner)$
$\quad\quad\quad \rightarrow \text{UNCHANGED } (localCaps)$
we cannot be the owner of the cap per our invariants $\rightarrow$ set remote ancs & descs to false because we won't need them
$\quad\quad \square\text{OTHER } \rightarrow localCaps' = [localCaps \text{ EXCEPT}$
$\quad\quad\quad\quad ![c] = localCaps[c]$
$\quad\quad\quad\quad\quad \cup \{[Head(messageQueues[c]).cap \text{ EXCEPT}$
$\quad\quad\quad\quad\quad\quad !.remote\_copies = \text{TRUE},$
$\quad\quad\quad\quad\quad\quad !.remote\_descs = \text{FALSE},$
$\quad\quad\quad\quad\quad\quad !.remote\_ancs = \text{FALSE}]\}])$
$\quad \land DequeueMsg(c)$
$\quad \land \text{UNCHANGED } (\langle deleteHandlers,\ requestHandlers\rangle)$

$ReceiveDeletionNotice(c) \triangleq$
$\text{LET } cap \triangleq Head(messageQueues[c]).cap \text{IN}$

```
LET copies ≜ {cap2 ∈ localCaps[c] : CapIsCopy(cap2, cap)}IN
    ∧ ∀ cap2 ∈ copies : cap2.owner ≠ c
    ∧ (CASE copies ≠ {}
        → ∧ BroadcastDequeue([type ↦ "DeletionAck",
                              sender ↦ c,
                              transfer ↦ TRUE,
                              cap ↦ cap], c)
          MUST EITHER SET REMOTE_COPIES TO TRUE (OR ALTERNATIVELY WAIT
          FOR A DELETION ACK OF OTHER NODE)
          OW WE MIGHT LOCALLY DELETE AND RETYPE THE CAP BEFORE THE
          FORMER OWNER'S CAP IS DELETED!
          ∧ localCaps' = [
                localCaps EXCEPT
                    ![c] = (localCaps[c] \ copies)
                           ∪ {[copy EXCEPT
                                  !.owner = c,
                                  !.remote_descs = cap.remote_descs,
                                  !.remote_copies = TRUE
                              ] : copy ∈ copies}]
        □ OTHER → ∧ BroadcastDequeue([type ↦ "DeletionAck",
                                      sender ↦ c,
                                      transfer ↦ FALSE,
                                      cap ↦ cap], c)
                  ∧ UNCHANGED (localCaps))
    ∧ UNCHANGED (⟨deleteHandlers, requestHandlers⟩)
MarkRevoke(c, cap, init) ≜
    LET owned_copies ≜ {cap2 ∈ localCaps[c] :
                          ∧ CapIsCopy(cap2, cap)
                          ∧ cap2.owner = c}IN
    LET owned_descendants ≜ {cap2 ∈ localCaps[c] :
                               ∧ CapContained(cap2, cap)
                               ∧ cap2.owner = c} \ owned_copiesIN
    LET remote_descendants_copies ≜
            {cap2 ∈ localCaps[c] :
                ∧ CapContained(cap2, cap)
                ∧ cap2.owner ≠ c}IN
    CASE init ∧ owned_copies ≠ {} →
        localCaps' = [localCaps EXCEPT ![c] =
          silently remove all remote_descendants_copies
         (localCaps[c] \ (owned_descendants ∪ remote_descendants_copies))
          mark all descendants as in_delete, remove remote copies & descendants
         ∪ {[d EXCEPT !.in_delete = TRUE,
                      !.remote_copies = FALSE,
                      !.remote_descs = FALSE] : d ∈ owned_descendants}]
```

```
    □¬init ∧ owned_copies = {} →
        LET parents ≜ getParent(localCaps[c], cap)IN
        localCaps' = [localCaps EXCEPT ![c] =
        (localCaps[c] \ (owned_descendants ∪ remote_descendants_copies ∪ parents))
         mark all descendants and copies as in_delete, remove remote copies & descendants
         ∪ {[d EXCEPT !.in_delete = TRUE,
                      !.remote_copies = FALSE,
                      !.remote_descs = FALSE] : d ∈ (owned_descendants)}
         ∪ {[p EXCEPT !.remote_descs = TRUE] : p ∈ parents}]
      fail if owned_copies is not of expected form

ReceiveRevokeMark(c) ≜
  LET cap ≜ Head(messageQueues[c]).capIN
  ∧ ∀ cap2 ∈ localCaps[c] :
     CapIsCopy(cap2, cap) ⇒ cap2.owner ≠ c
  ∧ MarkRevoke(c, cap, FALSE)
  ∧ BroadcastDequeue([type ↦ "RevokeMarkDone",
                      sender ↦ c], c)
  ∧ deleteHandlers' = [deleteHandlers EXCEPT ![c] =
                        [
                         deleteHandlers[c] EXCEPT
                         !.pending_resumes = deleteHandlers[c].pending_resumes + 1
                        ]
                       ]
  ∧ UNCHANGED (requestHandlers)

RevokeMarkStep(c) ≜
  LET rh ≜ requestHandlers[c]IN
  ∧ rh.state = "REVOKE_MARK"
  ∧ ¬rh.ack_received
  ∧ MarkRevoke(c, rh.cap, TRUE)
  ∧ requestHandlers' = [requestHandlers EXCEPT
                        ![c] = [state ↦ "REVOKE_SWEEP",
                                cap ↦ rh.cap,
                                retype_cap ↦ [owner ↦ "None"],
                                delete_done ↦ FALSE,
                                ack_received ↦ FALSE
                               ]]
  ∧ BroadcastDequeue([type ↦ "RevokeSweep", sender ↦ c], c)
  ∧ deleteHandlers' = [deleteHandlers EXCEPT ![c] = [
                       deleteHandlers[c] EXCEPT
                         !.inform_request_handler = TRUE,
                         !.pending_resumes = deleteHandlers[c].pending_resumes − 1]]
```

```
ReceiveRevokeSweep(c) ≜
 LET dh ≜ deleteHandlers[c] IN
  ∧ DequeueMsg(c)
  ∧ deleteHandlers' = [
        deleteHandlers EXCEPT ![c] = [
                    deleteHandlers[c] EXCEPT
                       !.pending_resumes = dh.pending_resumes − 1,
                       !.msgs = dh.msgs ∪ {[type ↦ "RevokeSweepDone",
                                            sender ↦ c]}]]
  ∧ UNCHANGED (⟨localCaps, requestHandlers⟩)

ReceiveDeletionAck(c) ≜
    LET copies ≜ {cap ∈ localCaps[c] : CapIsCopy(cap, deleteHandlers[c].cap)} IN
      copy should still be there!
      but, if revoke mark happened in the mean time, the remote_copies & descendants may have been unset
      thus, perform simple deletion on copies / copies with remote_copies unset
     ∧ Cardinality(copies) = 1
     ∧ DequeueMsg(c)
     ∧ deleteHandlers' = [deleteHandlers EXCEPT ![c] = [deleteHandlers[c] EXCEPT
                                                          !.state = "READY",
                                                          !.cap = [owner ↦ "None"]]]
        transfer happened → this means that a remote copy exists, account for it when deleting!
     ∧ CASE Head(messageQueues[c]).transfer → ∀ copy ∈ copies : SimpleDelete(copy, c)
        transfer did not happen → remote_copy already deleted, so unset flag
       □OTHER → ∀ copy ∈ copies : SimpleDelete([copy EXCEPT !.remote_copies = FALSE], c)
     ∧ UNCHANGED (requestHandlers)

ReceiveRevokeSweepDone(c) ≜
  LET rh ≜ requestHandlers[c] IN
  LET copies ≜ {cap ∈ localCaps[c] : CapIsCopy(cap, rh.cap)} IN
   ∧ rh.state = "REVOKE_SWEEP"
   ∧ ¬rh.ack_received
   ∧ DequeueMsg(c)
   ∧ requestHandlers' = [requestHandlers EXCEPT ![c] =
                                [requestHandlers[c] EXCEPT
                                    !.ack_received = TRUE]]
   ∧ localCaps' = [localCaps EXCEPT ![c] =
                          (localCaps[c] \ copies)
                          ∪ {[copy EXCEPT
                                !.locked = FALSE,
                                !.remote_descs = FALSE,
                                !.remote_copies = FALSE] : copy ∈ copies}]
   ∧ UNCHANGED (⟨deleteHandlers⟩)
```

```
ReceiveCanRetypeMsg(c) ≜
 LET msg ≜ Head(messageQueues[c])IN
 LET copies ≜ {cap ∈ localCaps[c] : CapIsCopy(cap, msg.cap)}IN
  if we own the cap and it's not "busy", we perform the retype ourselves too to break symmetry
 LET ownership_transfer_retype ≜
              ∧ copies ≠ {}
              ∧ (∀ copy ∈ copies :
                   ∧ copy.owner = c
                   ∧ ¬copy.in_delete
                   ∧ ¬copy.locked)
              ∧ CanRetype(msg.cap, msg.retype_cap, c)IN
 ∧ (CASE ownership_transfer_retype
     we lock the resulting retyped capabilities because of their ownership transfer
     → ( ∧ localCaps′ = [localCaps EXCEPT ![c] =
                              localCaps[c] ∪ {[msg.retype_cap EXCEPT
                                                !.locked = TRUE,
                                                !.remote_copies = TRUE,
                                                !.owner = c]}]
         ∧ BroadcastDequeue([type ↦ "CanRetypeAnswer",
                             sender ↦ c,
                             answer ↦ TRUE,
                             transfer ↦ TRUE], c)
       )
     if we don't keep the retype because the cap is locked / in delete (means that we don't
     perform any retypes for ourselves, so that's safe) we must still update the remote_descs
     □OTHER → ( ∧ localCaps′ = [localCaps EXCEPT ![c] =
                              (localCaps[c] \ copies)
                              ∪ {[copy EXCEPT
                                    !.remote_descs =
                                       ∨ copy.remote_descs
                                          no need to set remote descs on remotely owned capability
                                       ∨ ∧ CanRetype(msg.cap, msg.retype_cap, c)
                                         ∧ copy.owner = c
                                  ] : copy ∈ copies}]
                ∧ BroadcastDequeue([type ↦ "CanRetypeAnswer",
                                    sender ↦ c,
                                    answer ↦ CanRetype(msg.cap, msg.retype_cap, c),
                                    transfer ↦ FALSE], c)
              )
     )
 ∧ UNCHANGED (⟨requestHandlers, deleteHandlers⟩)

ReceiveOwnershipTransfer(c) ≜
   LET cap ≜ Head(messageQueues[c]).capIN
```

```
    LET copies ≜ {cap2 ∈ localCaps[c] : CapIsCopy(cap, cap2)}IN
      ∧ BroadcastDequeue([type ↦ "OwnershipTransferAck",
                          sender ↦ c,
                          cap ↦ cap], c)
      ∧ localCaps' = [localCaps EXCEPT ![c] =
                        (localCaps[c] \ copies)
                        ∪ {[cap EXCEPT
                              !.owner = c,
                              !.remote_copies = TRUE,
                              !.locked = FALSE,
                              !.in_delete = FALSE]}]
      ∧ UNCHANGED (⟨requestHandlers, deleteHandlers⟩)

TransferStep(c) ≜
    LET rh ≜ requestHandlers[c]IN
    LET copies ≜ {cap ∈ localCaps[c] : CapIsCopy(rh.cap, cap)}IN
        ∧ rh.state = "TRANSFER"
        ∧ ∀ copy ∈ copies : copy.locked
        cap was marked for deletion in the mean time
        ∧ (CASE ∃ copy ∈ copies : copy.in_delete →
              SimpleDelete([rh.cap EXCEPT !.remote_copies = TRUE], c)
          □OTHER →
            localCaps' = [localCaps EXCEPT ![c] =
                            (localCaps[c] \ copies)
                            ∪ {[copy EXCEPT
                                  !.owner = Head(messageQueues[c]).sender,
                                  !.remote_copies = TRUE,
                                  !.remote_descs = FALSE,
                                  !.locked = FALSE
                               ] : copy ∈ copies}])
        ∧ requestHandlers' = [requestHandlers EXCEPT ![c] =
                                [state ↦ "IDLE",
                                 cap ↦ [owner ↦ "None"],
                                 retype_cap ↦ [owner ↦ "None"],
                                 delete_done ↦ FALSE,
                                 ack_received ↦ FALSE]]
        ∧ DequeueMsg(c)
        ∧ UNCHANGED (deleteHandlers)

RetypeQueryingStep(c) ≜
    LET rh ≜ requestHandlers[c]IN
    LET copies ≜ {cap ∈ localCaps[c] : CapIsCopy(cap, requestHandlers[c].cap)}IN
    LET canRetype ≜ ∧ Head(messageQueues[c]).answer
                    ∧ copies ≠ {}
                    ∧ ∀ copy ∈ copies : ( ∧ CanRetype(copy, rh.retype_cap, c)
```

$\wedge \neg copy.in_delete)$IN
$\wedge\ rh.state = $ "RETYPE_QUERYING"
$\wedge \neg rh.ack\_received$
$\wedge\ requestHandlers' = [requestHandlers$ EXCEPT $![c] =$
$[state \mapsto$ "IDLE",
$cap \mapsto [owner \mapsto$ "None"],
$retype\_cap \mapsto [owner \mapsto$ "None"],
$delete\_done \mapsto$ FALSE,
$ack\_received \mapsto$ FALSE$]]$
$\wedge$ (CASE $Head(messageQueues[c]).transfer$
$\rightarrow BroadcastDequeue([type \mapsto$ "RetypeTransferAck",
$answer \mapsto canRetype,$
$cap \mapsto rh.retype\_cap,$
$sender \mapsto c],\ c)$
$\square$OTHER $\rightarrow DequeueMsg(c))$
$\wedge$ UNCHANGED $(\langle deleteHandlers\rangle)$
$\wedge$ (CASE $canRetype$
$\rightarrow localCaps' = [localCaps$ EXCEPT $![c] =$
$localCaps[c]$
$\cup \{[rh.retype\_cap$ EXCEPT
$!.remote\_copies = Head(messageQueues[c]).transfer]\}]$
$\square$OTHER $\rightarrow$ UNCHANGED $(localCaps))$

$ReceiveRetypeTransferAck(c) \triangleq$
LET $msg \triangleq Head(messageQueues[c])$IN
LET $copies \triangleq \{cap \in localCaps[c] : CapIsCopy(cap,\ msg.cap)\}$IN
$\wedge\ DequeueMsg(c)$
$\wedge$ UNCHANGED $(\langle requestHandlers,\ deleteHandlers\rangle)$
$\wedge$ CASE $msg.answer \wedge \forall\, copy \in copies : \neg copy.in\_delete$
$\rightarrow localCaps' = [localCaps$ EXCEPT $![c] =$
$(localCaps[c] \setminus copies)$
$\cup \{[copy$ EXCEPT
$!.locked =$ FALSE,
$!.owner = msg.sender$
$] : copy \in copies\}]$
$\square msg.answer \wedge \exists\, copy \in copies : copy.in\_delete$
$\rightarrow localCaps' = [localCaps$ EXCEPT $![c] = (localCaps[c] \setminus copies)]$
$\square$OTHER $\rightarrow localCaps' = [localCaps$ EXCEPT $![c] = (localCaps[c] \setminus copies)$
$\cup \{[copy$ EXCEPT
$!.locked =$ FALSE
$] : copy \in copies\}]$

$ReceiveMsg(c) \triangleq$
$\wedge\ Len(messageQueues[c]) > 0$
$\wedge$ CASE $Head(messageQueues[c]).type =$ "SendCapMsg" $\rightarrow ReceiveCapMsg(c)$

$\Box Head(messageQueues[c]).type = \text{“DeletionNotice”} \rightarrow ReceiveDeletionNotice(c)$
$\Box Head(messageQueues[c]).type = \text{“DeletionAck”} \rightarrow ReceiveDeletionAck(c)$
$\Box Head(messageQueues[c]).type = \text{“RevokeMark”} \rightarrow ReceiveRevokeMark(c)$
$\Box Head(messageQueues[c]).type = \text{“RevokeMarkDone”} \rightarrow RevokeMarkStep(c)$
$\Box Head(messageQueues[c]).type = \text{“RevokeSweep”} \rightarrow ReceiveRevokeSweep(c)$
$\Box Head(messageQueues[c]).type = \text{“RevokeSweepDone”} \rightarrow ReceiveRevokeSweepDone(c)$
$\Box Head(messageQueues[c]).type = \text{“CanRetypeMsg”} \rightarrow ReceiveCanRetypeMsg(c)$
$\Box Head(messageQueues[c]).type = \text{“CanRetypeAnswer”} \rightarrow RetypeQueryingStep(c)$
$\Box Head(messageQueues[c]).type = \text{“RetypeTransferAck”} \rightarrow ReceiveRetypeTransferAck(c)$
$\Box Head(messageQueues[c]).type = \text{“OwnershipTransfer”} \rightarrow ReceiveOwnershipTransfer(c)$
$\Box Head(messageQueues[c]).type = \text{“OwnershipTransferAck”} \rightarrow TransferStep(c)$
$\Box \text{OTHER} \rightarrow \text{FALSE}$

$RevokeSweepStep(c) \triangleq$
$\quad \wedge requestHandlers' = [requestHandlers \text{ EXCEPT } ![c] =$
$\qquad\qquad [state \mapsto \text{“IDLE”},$
$\qquad\qquad cap \mapsto [owner \mapsto \text{“None”}],$
$\qquad\qquad retype\_cap \mapsto [owner \mapsto \text{“None”}],$
$\qquad\qquad delete\_done \mapsto \text{FALSE},$
$\qquad\qquad ack\_received \mapsto \text{FALSE}]]$
$\quad \wedge \text{UNCHANGED } (\langle messageQueues, deleteHandlers, localCaps \rangle)$

$DeleteDone(c) \triangleq$
$\quad \wedge requestHandlers' = [requestHandlers \text{ EXCEPT } ![c] =$
$\qquad\qquad [state \mapsto \text{“IDLE”},$
$\qquad\qquad cap \mapsto [owner \mapsto \text{“None”}],$
$\qquad\qquad retype\_cap \mapsto [owner \mapsto \text{“None”}],$
$\qquad\qquad delete\_done \mapsto \text{FALSE},$
$\qquad\qquad ack\_received \mapsto \text{FALSE}]]$
$\quad \wedge \text{UNCHANGED } (\langle messageQueues, localCaps, deleteHandlers \rangle)$

$ProcessRequest(c) \triangleq \text{LET } rh \triangleq requestHandlers[c] \text{IN}$
$\qquad \text{CASE } \wedge rh.state = \text{“REVOKE\_SWEEP”}$
$\qquad\qquad\quad \wedge rh.ack\_received$
$\qquad\qquad\quad \wedge rh.delete\_done$
$\qquad\qquad\quad \rightarrow RevokeSweepStep(c)$
$\qquad\quad \Box \wedge rh.state = \text{“DELETE\_PROCESSING”}$
$\qquad\qquad \wedge rh.delete\_done$
$\qquad\qquad\quad \rightarrow DeleteDone(c)$
$\qquad\quad \Box \text{OTHER} \rightarrow \text{FALSE}$

$DeleteStep(c) \triangleq$
$\quad \text{LET } dH \triangleq deleteHandlers[c] \text{IN}$

```
    ∃ cap ∈ localCaps[c] :
     ∧ cap.in_delete
     ∧ ¬cap.locked
     ∧ dH.state = "READY"
     ∧ dH.pending_resumes = 0
     ∧ CASE ¬cap.remote_copies ∨ cap.owner ≠ c
               → ∧ SimpleDelete(cap, c)
                 ∧ UNCHANGED (⟨messageQueues, deleteHandlers, requestHandlers⟩)
       □OTHER  → ∧ UNCHANGED (⟨localCaps, requestHandlers⟩)
                 ∧ Broadcast([type ↦ "DeletionNotice",
                              cap ↦ [cap EXCEPT
                                     !.remote_descs = HasLocalDescs(cap, localCaps[c])],
                              sender ↦ c], c)
                 ∧ deleteHandlers′ = [deleteHandlers EXCEPT
                                      ![c] = [deleteHandlers[c] EXCEPT
                                              !.state = "COORDINATING",
                                              !.cap = cap]]

DeleteComplete(c) ≜
    LET dH ≜ deleteHandlers[c] IN
     ∧ ¬∃ cap ∈ localCaps[c] : cap.in_delete
     ∧ dH.pending_resumes = 0
     ∧ deleteHandlers′ = [deleteHandlers EXCEPT ![c] =
                          [deleteHandlers[c] EXCEPT
                           !.inform_request_handler = FALSE,
                           !.msgs = {}]]
     ∧ requestHandlers′ = [requestHandlers EXCEPT ![c] =
                           [requestHandlers[c] EXCEPT
                            !.delete_done =
                              ∨ deleteHandlers[c].inform_request_handler
                              ∨ requestHandlers[c].delete_done]]
     ∧ messageQueues′ = [core ∈ Cores ↦
                         IF core = c
                          THEN messageQueues[core]
                          ELSE messageQueues[core] ∘ SeqFromSet(deleteHandlers[c].msgs)]
     ∧ UNCHANGED (localCaps)

Revoke(c) ≜
    LET rh ≜ requestHandlers[c] IN
    ∃ cap ∈ localCaps[c] : cap.owner = c
     ∧ requestHandlers[c].state = "IDLE"
     ∧ ¬cap.in_delete
     ∧ ¬cap.locked
     ∧  Broadcast([type ↦ "RevokeMark",
```

```
                    cap ↦ cap,
                    sender ↦ c], c)
    ∧  requestHandlers′ =
          [requestHandlers EXCEPT ![c] =
                   [state ↦ "REVOKE_MARK",
                    cap ↦ cap,
                    retype_cap ↦ [owner ↦ "None"],
                    delete_done ↦ FALSE,
                    ack_received ↦ FALSE]]
     lock revocation target
    ∧ localCaps′ = [localCaps EXCEPT ![c] =
                              (localCaps[c] \ {cap})
                              ∪ {[cap EXCEPT
                                    !.locked = TRUE]}]
    ∧ deleteHandlers′ = [deleteHandlers EXCEPT ![c] =
                                 [deleteHandlers[c] EXCEPT
                                       !.pending_resumes = deleteHandlers[c].pending_resumes + 1]]


Delete(c) ≜
   ∃ cap ∈ localCaps[c] :
   LET rh ≜ requestHandlers[c]IN
    ∧ CASE   ∧  ∨ cap.owner ≠ c
                ∨  ∧ ¬cap.remote_copies
                   ensure that root caps are not deleted
                   ∧ ¬∃ root_cap ∈ UNION {BaseCaps[core] : core ∈ Cores} : CapIsCopy(root_cap, cap)
             ∧ ¬cap.locked
             ∧ ¬cap.in_delete
           →
             ∧ SimpleDelete(cap, c)
             ∧ UNCHANGED (⟨messageQueues, deleteHandlers, requestHandlers⟩)
      □ ∧ cap.owner = c
        ∧ cap.remote_copies
        ∧ ¬cap.locked
        ∧ ¬cap.in_delete
        ∧ ¬∃ root_cap ∈ UNION {BaseCaps[core] : core ∈ Cores} : CapIsCopy(root_cap, cap)
        ∧ requestHandlers[c].state = "IDLE"
        →
        ∧ requestHandlers′ = [requestHandlers EXCEPT ![c] =
                                    [state ↦ "DELETE_PROCESSING",
                                     cap ↦ cap,
                                     retype_cap ↦ [owner ↦ "None"],
                                     delete_done ↦ FALSE,
                                     ack_received ↦ FALSE]]
```

$\land\ deleteHandlers' = [deleteHandlers \text{ EXCEPT } ![c] =$
$\qquad [deleteHandlers[c] \text{ EXCEPT}$
$\qquad\quad !.inform\_request\_handler = \text{TRUE}]]$
$\land\ localCaps' = [localCaps \text{ EXCEPT } ![c] =$
$\qquad (localCaps[c] \setminus \{cap\})$
$\qquad \cup \{[cap \text{ EXCEPT } !.in\_delete = \text{TRUE}]\}]$
$\land\ \text{UNCHANGED } (\langle messageQueues \rangle)$
$\Box\text{OTHER } \rightarrow \text{FALSE}$

$Retype(c) \triangleq$
$\text{LET } rh \triangleq requestHandlers[c] \text{IN}$
$\exists\, cap \in localCaps[c] :$
$\exists\, newstart \in cap.start \,..\, (cap.end - 1) :$
$\exists\, newend \in (newstart + 1) \,..\, (cap.end) :$
$\text{LET } retype\_cap \triangleq [$
$\quad start \mapsto newstart,$
$\quad end \mapsto newend,\ owner \mapsto c,$
$\quad remote\_copies \mapsto \text{FALSE},$
$\quad remote\_ancs \mapsto \text{FALSE},$
$\quad remote\_descs \mapsto \text{FALSE},$
$\quad in\_delete \mapsto \text{FALSE},$
$\quad locked \mapsto \text{FALSE}]\text{IN}$
$\land\ \ CanRetype(cap,\ retype\_cap,\ c)$
$\land\ \ (\text{CASE } cap.owner = c \land \neg cap.remote\_descs$
$\qquad \rightarrow LocalRetype(c,\ cap,\ retype\_cap)$
$\qquad \Box(\land (cap.owner \neq c\ \ \lor cap.remote\_descs)$
$\qquad\ \land requestHandlers[c].state = \text{“IDLE”})$
$\qquad\ \rightarrow$
$\qquad (\land Broadcast([type \mapsto \text{“CanRetypeMsg”},$
$\qquad\qquad\qquad cap \mapsto cap,$
$\qquad\qquad\qquad retype\_cap \mapsto retype\_cap,$
$\qquad\qquad\qquad sender \mapsto c],\ c)$
$\qquad\ \land requestHandlers' = [requestHandlers \text{ EXCEPT } ![c] =$
$\qquad\qquad\qquad\qquad [state \mapsto \text{“RETYPE\_QUERYING”},$
$\qquad\qquad\qquad\qquad\ cap \mapsto cap,$
$\qquad\qquad\qquad\qquad\ retype\_cap \mapsto retype\_cap,$
$\qquad\qquad\qquad\qquad\ delete\_done \mapsto \text{FALSE},$
$\qquad\qquad\qquad\qquad\ ack\_received \mapsto \text{FALSE}]]$
$\qquad\ \land \text{UNCHANGED } (\langle deleteHandlers,\ localCaps \rangle))$
$\qquad \Box\text{OTHER } \rightarrow \text{FALSE}\quad)$

$TransferCapOwnership(c) \triangleq$
$\text{LET } rh \triangleq requestHandlers[c] \text{IN}$

$\exists\, cap \in localCaps[c] :$
$\land\ cap.owner = c$
$\land\ \neg cap.in\_delete$
$\land\ \neg cap.locked$
$\land\ rh.state = \text{“IDLE”}$
$\land\ Broadcast([type \mapsto \text{“OwnershipTransfer”},$
$cap \mapsto [cap \text{ EXCEPT } !.remote\_descs = HasLocalDescs(cap,\ localCaps[c])],$
$sender \mapsto c],\ c)$
$\land\ localCaps' = [localCaps \text{ EXCEPT } ![c] =$
$(localCaps[c] \setminus \{cap\})$
$\cup \{[cap \text{ EXCEPT}$
$!.locked = \text{TRUE}]\}]$
$\land\ requestHandlers' = [requestHandlers \text{ EXCEPT } ![c] =$
$[state \mapsto \text{“TRANSFER”},$
$cap \mapsto cap,$
$retype\_cap \mapsto [owner \mapsto \text{“None”}],$
$delete\_done \mapsto \text{FALSE},$
$ack\_received \mapsto \text{FALSE}]]$
$\land\ \text{UNCHANGED}\ (deleteHandlers)$

$CapSystemStep \triangleq \exists\, c \in Cores :$
$\lor\ SendCap(c)$
$\lor\ ReceiveMsg(c)$
$\lor\ DeleteStep(c)$
$\lor\ ProcessRequest(c)$
$\lor\ Revoke(c)$
$\lor\ DeleteComplete(c)$
$\lor\ Delete(c)$
$\lor\ Retype(c)$
$\lor\ TransferCapOwnership(c)$

* Modification History
* Last modified *Thu Jun* 02 17:23:22 *CEST* 2022 by *jasmin*
* Created *Wed* May 18 10:54:09 *CEST* 2022 by *jasmin*